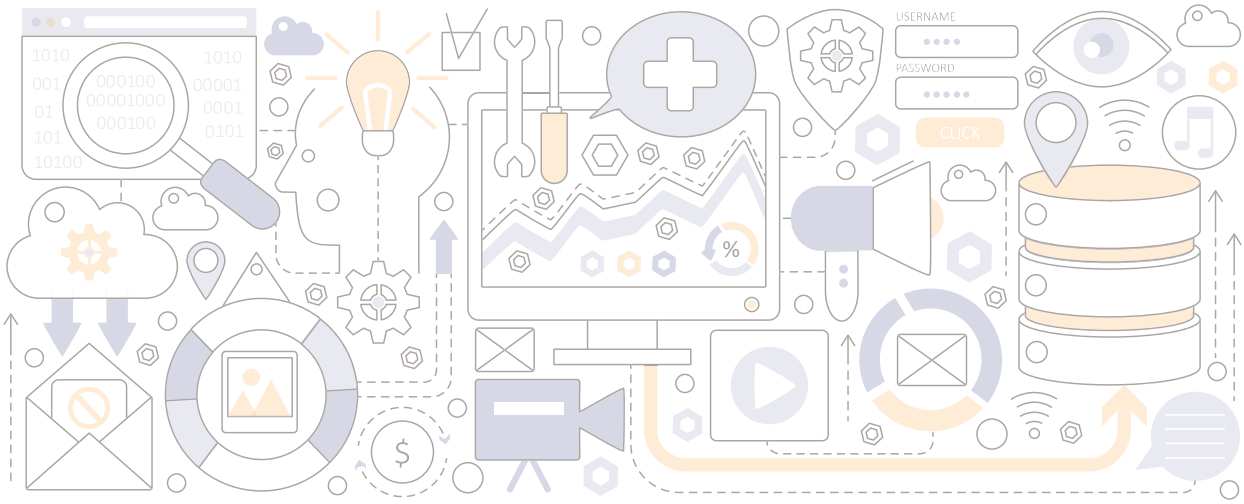
Deep Learning Models and Applications

深度学习

模型及应用详解

张若非　付强　高斌　张耿豪　叶挺　著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书作者都是微软人工智能及研究院的研究人员和应用科学家，具有深厚的机器学习背景，在一线针对产品需求和支持的场景进行了大量的深度学习模型及算法的研究和开发，在模型设计、训练、评估、部署、推理优化等模型开发全生命周期积累了丰富的经验。

本书分为 4 部分，共 13 章。其中第 1 部分（第 1、2 章）简要介绍了深度学习的现状、概念和实现工具。第 2 部分（第 3～5 章）以具体的实际应用展示基于深度学习技术进行工程实践和开发的流程和技巧。第 3 部分（第 6～12 章）介绍了学术界和工业界最新的高阶深度学习模型的实现和应用。第 4 部分（第 13 章）介绍了深度学习领域的一些前沿研究方向，并对深度学习的未来发展进行展望。

本书面向的读者是希望学习和运用深度学习模型到具体应用场景的企业工程师、科研院所的学生和科研人员。读者学习本书的目的是了解深度学习模型和算法基础后，快速部署到自己的工作领域，并取得落地成果。

图书在版编目（CIP）数据

深度学习模型及应用详解 / 张若非等著. —北京：电子工业出版社，2019.9
ISBN 978-7-121-37126-4

Ⅰ. ①深… Ⅱ. ①张… Ⅲ. ①机器学习－研究 Ⅳ. ①TP181

中国版本图书馆 CIP 数据核字（2019）第 150112 号

策划编辑：张慧敏
责任编辑：石　倩
文字编辑：王　静　吴宏伟
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：720×1000　1/16　印张：17.25　字数：303.6 千字
版　　次：2019 年 9 月第 1 版
印　　次：2019 年 9 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

Deep
Learning
Models and Applications

序言
Preface

机器学习从诞生那天起，其模型的设计和演变就和相关应用是密切结合的。近十年来，伴随着大数据和高性能计算硬件的迅猛发展，机器学习里面的深度学习异军突起，为人工智能领域中的诸多应用提供了核心算法模型。在计算机、电子工程、数学等相关学科，越来越多的学生加入到深度学习的领域进行深造；在信息技术、互联网等相关领域，越来越多的工程师运用深度学习技术来提升产品性能；在金融、医疗、汽车等传统领域，越来越多的从业者正积极地设法引入深度学习技术，从而掀起新一轮的技术革命。

在如此宏大的技术普及和发展的背景下，亟需一本介绍深度学习技术实践的图书。目前在市面上可以看到的与深度学习相关的书籍有数十本，各具特色。有的侧重于模型的推导和解释，有的侧重于算法的实现，而本书的特点是侧重于深度学习技术的落地。我在微软访问时与本书的几位作者有过多次接触，后来在一些学术会议上也分别有过几次讨论。他们都是毕业于国内外著名学府的博士和硕士，在机器学习领域有着多年的研究、开发经验，在国际顶级期刊和会议上发表了数十篇论文，并在微软的产品部门从事一线产品的工程实践。如今，看到他们在工作之余，将深度学习中的一些重要模型在实践中的应用加以整理并总结成

书，甚感欣慰。这将给深度学习技术在工程实践中的普及带来非常重要的促进作用。

读完本书，我的感受是，内容全面翔实、条理清晰：从最初的感知机模型到最近的对抗生成网络和深度强化学习，作者都进行了详尽的介绍；从模型的原理、应用的细节到计算框架的选择，作者都进行了全面阐述。相信认真读完本书的读者，一方面会对深度学习的概貌有全局性的了解，既见树木，更见森林；另一方面，可以对一些深度学习的实际应用问题厘清思路，甚至在本书中直接找到解决方案。不论是深度学习领域的初学者，还是具有经验的工程实践者，都可以从本书中获得启示，做到开卷有益！

最后，祝贺这本书的出版，并希望它成为深度学习实践者的案头手册。

杨强

香港科技大学计算机科学与工程学系讲座教授

国际人工智能联合会（IJCAI）理事会主席 （2017—2019）

深圳前海微众银行首席 AI 官

Deep
Learning
Models and Applications

前言
Introduction

人工智能热潮

人工智能已成为炙手可热的名词和话题，其范围和影响力已经超越了学术研究和产业科技研究，成为一个社会性热点。人工智能被广泛认为是具有颠覆性的战略技术领域，对未来的世界发展和社会进步有重大影响，是建设创新型国家和世界科技强国的重要支撑，各国也相继发布关于人工智能的国家发展战略和规划。2017 年 7 月，国务院发布了《新一代人工智能发展规划》的精神和部署，对我国在人工智能基础理论研究、核心技术、模型和算法、软硬件支撑平台、生态系统建设等方面规划了蓝图。这进一步激起了学术界、工业界、政府等社会各方面人士对人工智能的关注、学习、研究和开发。

人工智能的引爆在很大程度上源自深度学习技术的突破，包括语音、视觉和决策规划等领域。比如，2012 年基于深度学习模型的 AlexNet 首次夺得 ImageNet 大规模视觉识别竞赛（ILSVRC）的冠军，之后所有这个竞赛的优胜模型全部是基于深度学习的。2016 年 3 月，DeepMind 基于深度强化学习模型的围棋程序 AlphaGo 战胜围棋世界冠军李世石引起轰动，获得广泛关注，也敲响了人工智能在社会上热潮开启的晨钟。随即各个领域学习深度学习的兴趣日渐高

涨，深度学习模型、算法、框架、工具、软硬件加速器等的研究和开发也如火如荼，在研究、科技和商业领域都获得了迅猛发展。

机器学习发展路径

大部分深度学习是基于深层神经元网络的模型，属于机器学习的一种。要学习深度学习，有必要了解机器学习的发展历程。机器学习领域有很多流派，例如，强调“推理、知识、学习”的人工智能派和强调应用统计学的统计学派。

机器学习从提出、研究到发展，至今有六十多年了。这中间的研究人员有过很多方法论的尝试，让机器能够像人一样思考、判断、预测。在这个过程中，不同的时期有不同的方法流行，而在另外一个时期又失去了吸引力，但后来可能又获得新生。机器学习的发展过程可以用波浪式前进、螺旋式上升来概括。这也和每个时期的技术条件、研究水平、人们的认知水平，尤其是对人类大脑的了解，以及社会整体文明进步水平有关。

20 世纪 80 年代初，机器学习研究主要集中在对知识的描述和表达、存储，以及用知识库进行推理方面。其中，用符号表示人工智能（symbolic AI）比较流行，它集中在高层次的、人类可理解的，对问题、逻辑和搜索的符号表达上，以及基于其上的规则系统的构建，最具代表性的是专家系统。但是专家系统的功能和性能远远达不到人们的期望，而且专家系统也没有数学理论的支持，很难证明这种方法论的稳定性和正确性。

20 世纪 90 年代后期，随着 Vapnik 统计学习理论的研究成熟，迎来了统计机器学习的黄金时期。此时出现众多的统计学习模型，比如贝叶斯网络、朴素贝叶斯、最大熵、支持向量机（Support Vector Machine，SVM）、决策树（decision tree），普遍使用的梯度提升决策树（GBDT）、随机森林（random forest）、矩阵分解模型等，可以说是百花齐放，在各种分类、回归、聚簇问题上的准确性明显提高。因此，在搜索、广告、推荐等大量的互联网场景下获得了广泛的应用。

统计机器学习模型获得成功的一个重要原因是它有稳固的统计学和最优化等数学理论的支撑，为机器学习研究和学习能力的提高提供了理论上的保证和方向上的指导。机器学习模型不是一个黑盒子，而是基于严格的数学计算，这非常重要。在整个 21 世纪的第一个十年，都是统计机器学习的天下，但是这些统计机器学习模型往往需要领域专业人士和数据科学家做大量的特征工程（feature engineering）工作，设计有效的特征，才能输入模型，得到满意的效果。

在众多统计学习模型中，人工神经网络是一大类算法。人工神经网络的发展同样经历了高潮低谷的交替起伏。在深度学习兴起之前的约 20 年时间里，由于计算能力和数据量的限制，人工神经网络的有效训练和学习往往只能停留在浅层次的小规模神经网络上，限制了其学习性能。此外，人工神经网络学习得到的模型也缺乏直观的可解释性。这些因素使得人工神经网络逐渐失去了吸引力。

近年来，由于大数据的发展，大量可用数据产生，以及计算能力的不断提升，神经网络卷土重来。同时，改善的模型结构及训练算法的提高，使深度学习得到了爆发，尤其是直接应用在一些做特征工程非常困难的原始数据的场景下，性能有突破性的进展，包括语音识别、图像理解、自然语言处理 （NLP）、机器翻译等，都取得了显著的改进。

本书的初衷

人工智能热激发了大家对深度学习的学习兴趣，但是目前的一些深度学习书籍要么是面向学校和机器学习的研究人员，重理论、少实践且不够实用；要么只是对一些深度学习框架和工具的介绍和翻译，而没有比较全面的深度学习模型的讲解、具体应用的实例及实际使用中经验和注意事项的分享。本书的作者都是在微软人工智能及研究院的研究人员和应用科学家，具有深厚的机器学习背景，在一线针对产品需求和支持的场景进行了大量的深度学习模型及算法的研究和开发，在模型设计、训练、评估、部署、推理优化等模型开发全生命周期积累了丰

富的经验。

本书面向的读者是希望学习和运用深度学习模型到具体应用场景的企业工程师、科研院所的学生和科研人员。他们的目的不是找一本教科书从学术角度学习深度学习，像深度学习研究人员一样设计新的模型和算法，而是对深度学习模型和算法做一个基础了解后，快速部署到他们的工作领域，并取得落地成果。这正是我们写作本书的初衷和希望有所贡献的地方：让读者“打基础、读得懂、用得快、重实践、重应用”，重点是建立起分析问题、对问题形式化和应用深度学习建模、使用工具实现模型训练和推理、在实际中需要考虑的约束限制、进行取舍和工程调优等一系列的方法论，从而获得能举一反三解决新的问题的能力。

本书内容定位

基于这个写作目的，在讲解基础的前提下，侧重在实际应用中让读者快速掌握基于深度学习模型的系统开发，本书的内容覆盖以下几个部分。

第 1 部分（第 1、2 章）讲解深度学习的现状、概念和实现工具。

第 2 部分（第 3~5 章）介绍深度学习在自然语言处理、计算机视觉、预测等应用中常见模型的举例及实现，包括自然语言处理中的词嵌入向量模型；图像理解中普遍使用的卷积神经网络（CNN）及其在物体检测（object detection）方面的应用；应用于机器翻译的递归神经网络模型（RNN）和长短期记忆模型（LSTM）。

第 3 部分（第 6~12 章）介绍学术界和工业界最新的一些高阶深度学习模型和实现，以及它们在互联网搜索、广告、对话机器人、电商等领域的应用，包括：用于对话机器人的 DeepProbe 模型；用于单张照片产品识别和属性生成的 VPR 模型；用于信息检索和语义向量生成的 DeepIntent 模型；用于文本语义嵌入和匹配的 fastText 模型；生成对抗网络（GAN），以及在图像生成和自然语言处理中的应用；强化学习模型的模型结构、训练算法和应用。这些模型和实现都已经

应用在微软的众多产品中，并获得了很好的效果。

第 4 部分（第 13 章）是讨论及展望，包括模型在线推理的优化及硬件加速的实现等。最后对目前深度学习技术的局限性做出分析，对其发展方向和下一个浪潮进行展望。

本书学习建议

我们建议读者按顺序阅读本书第 1 部分和第 2 部分，它们是本书的基础，难度由浅入深。学习了第 1 部分深度学习理论和常用工具后，再学习第 2 部分常见的深度学习模型。在学习过程中要配合实践，使用介绍的工具在一些实验数据集和示例问题上完成深度学习模型的训练及性能评估，掌握模型开发流程中的数据准备、训练、调试超参、评估、部署等全部步骤。第 3 部分是进阶内容，讲解一些应用于计算机视觉、自然语言处理及理解和决策任务的高级深度学习模型，各章节之间相互独立，内容没有前后依托关系，读者可以根据自己的兴趣和背景选择阅读。深度学习是一门实验科学，要在理论基础上结合应用大量进行实践，才能真正掌握并解决实际问题。

深度学习技术发展迅猛，新的模型、算法、工具、流程不断涌现，在传统互联网领域及各个行业的应用层出不穷，新的问题、新的解决方案也持续被提出。本书难以对深度学习各个层面做出全面深入的描述，一些最新的模型和应用也许没有包括进来。本书如能对读者学习深度学习模型、算法、实践和应用有所帮助，并在实践中产生加速和推动作用，那就达到了我们的目的。

由于本书作者水平、理解能力、经验和表达能力所限，一些错误、不足之处在所难免，恳请各位读者指正。

张若非

2019 年 7 月于美国硅谷山景城

Deep Learning
Models and Applications

目录
Contents

Deep Learning Models and Applications

第 1 章
神经网络发展史

在最近几年，深度神经网络已经在模式识别和机器学习领域的多个大型算法竞赛中取得冠军。例如，在 2009 年的 ICDAR（文档分析与识别国际会议）中，LSTM 在没有利用任何先验的语言学知识的前提下夺得了手写文字识别（法语、阿拉伯语、波斯语）的冠军。LSTM 在 2013 年还打破了著名的 TIMIT 语音识别竞赛的纪录。此外，从 2012 年至今，每一年 ImageNet 图像识别竞赛的冠军均是 CNN（卷积神经网络）的某种变种。

除这些著名的机器学习算法竞赛外，在很多实际应用中，深度神经网络所达到的准确率相比传统机器学习算法也有了本质的飞跃。特别是，Google 提出的基于深度学习和强化学习算法的 AlphaGo 在 2017 年击败了世界围棋冠军，引起了广泛关注。深度学习的巨大成功引发了人工智能的又一次热潮。现在，基于深度学习的各种智能应用正在渗透到各行各业中。从基础的数据分析到业务的应用（如自动驾驶、机器人），越来越多的基于深度学习的应用场景正在呈爆炸式增长。本书主要结合搜索广告这个在机器学习领域中最成功、最典型的大型应用案例，对深度学习技术进行详细介绍。

最常见的神经网络是前馈神经网络。一个前馈神经网络包含 3 种节点，即输入节点、输出节点以及隐藏节点。其中，外部输入的特征向量通过输入节点被传递给神经网络，经过一层或多层隐藏节点的计算后，最终得到输出结果，并通过输出节点输出，其结构示意图如图 1.1 所示。具体来说，输入层以外部输入特征向量作为其输入；隐藏层及输出层中的每个节点，以与其相连的前层节点的输出，按照对应连接边的权重进行加权求和作为其输入，再经过一个非线性激活函数（activation function）得到其输出。

所谓神经网络的学习，就是要找到一组连接边的权重，使得神经网络的输出与我们的目标越来越接近。需要说明的是，神经网络的结构，包括网络层数、各层神经元个数、各层的神经元之间如何连接、非线性映射函数的挑选等，可以有极大的不同。神经网络的训练算法也有很多变种。我们需要根据实际问题的特点，

设计和挑选合适的网络结构及训练算法。

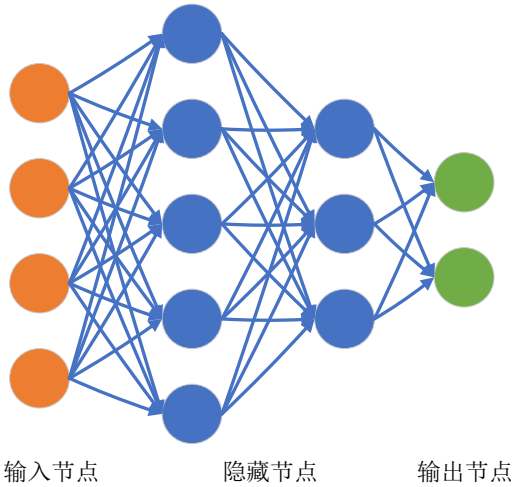


图 1.1　前馈神经网络结构示意图

理解一种技术的发展史，对于我们深入理解其精华与本质大有裨益。通过对技术发展史的总结与回顾，我们可以更加清晰地看到人类认知水平是如何不断提高的，前人曾经做过哪些尝试，哪些技术改进起到了决定性的推动作用，哪些想法是可以进一步借鉴的等。因此，本章我们会对神经网络的技术发展进行回顾和总结[1,2]。

1.1　神经网络的早期雏形

在最初研究神经网络时，人们最自然的想法就是，首先了解人脑的工作机制，然后用自动化手段进行模拟，从而实现人工智能。因此，神经网络的各种早期雏

1 SCHMIDHUBER J. Deep learning in neural networks: An overview. Neural Networks 2015[J] 61,85-117.

2 WANG H, RAJ B. On the Origin of Deep Learning. arXiv preprint arXiv:1702.07800v4,2017.

形在不同程度上体现了对人脑工作机制的猜想和模拟。需要指出的是，现在很多学者已经完全从计算的角度出发，而不再依赖于仿生学的研究结论了。因为很多人认为，就像飞机不是完全模拟鸟儿飞翔的原理一样，人工智能并不需要完全模拟人脑的结构和工作机制。

1.1.1 联结主义和 Hebb 学习规则

联结主义是著名的对人脑工作原理和结构的猜想。它是人工智能三大学派（符号主义、联结主义、行为主义）之一，又被称为仿生学派。具体来说，联结主义认为人脑是由大量的神经元组成的，这些神经元或神经元组对应着不同的事物或概念，这些神经元之间有着复杂的连接使其产生联系。但这些联系不是杂乱无章的，而是遵循一定规律的。例如，人脑会倾向于把那些在空间或时间上相邻的事务或事件连接在一起；两个事物如果经常一起发生，那么它们对应的神经元之间的连接强度就会比较大。

真正具体实现联结主义的早期算法之一是 Hebb 学习规则，它由 Donald O. Hebb 发明。Hebb 在他的著作中这样描述 Hebb 学习规则："当细胞 A 的轴突足够接近以激发细胞 B，并重复或持续地参与激发细胞 B 时，一些生长过程或代谢变化发生在一个或两个细胞中，使得 A（作为激发细胞 B 的众多细胞中的一个）激发细胞 B 的效率得以增强。"也就是说，临近细胞之间的连接强度会随着它们被同时激活的次数的增加而得到增强。Hebb 学习规则可以用下述公式进行精确表达。

$$\Delta w_i = \eta x_i y \tag{1}$$

其中，x_i表示神经元i的输入信号，w_i表示神经元i的突触连接强度，Δw_i表示w_i的变化。

1.1.2 Oja 学习规则及主分量分析

从 Hebb 学习规则的公式中不难看出，在 Hebb 学习规则下，神经元连接的权重会随着神经元之间被共同激活次数的增加而不断增加，并且那些频繁发生的事件对应的权重将呈指数增长。可以想象，由于公式（1）中的增量并不保证会趋于 0，因此，Hebb 学习规则在学习过程中可能会不收敛，这被称为 Hebb 学习规则的不稳定性。

Erkki Oja 针对 Hebb 学习规则的不稳定性进行了改进，提出了 Oja 学习规则。Oja 学习规则可以用下面的公式表示。

$$w_i^{t+1} = w_i^t + \eta y(x_i - y w_i^t) \tag{2}$$

公式中的符号意义同公式（1），t表示迭代次数的索引值。这种形式可以被理解为一种归一化处理，使得权重w_i不会随着迭代次数的增加而无限增大，从而克服了 Hebb 学习规则的权重不稳定性问题。从另一个角度看，它也可以被理解为用负反馈机制来抑制权重的无限增大。此外，Oja 进一步表明，他的学习规则相当于用在线更新的方式实现了主分量分析。

1.1.3 早期的神经元模型

MCP 是于 19 世纪 40 年代被提出的神经元模型。其发明者对神经元的反应过程加以推测和抽象，并用电路模拟神经元的工作原理。具体来说，他们将神经元的反应过程抽象为：神经元对其多个输入信号进行加权求和，形成总的输入信号，然后通过非线性激活函数（阈值函数）产生神经元的输出信号。其中，y表示输出值，x表示输入值，w表示权重，θ表示激活阈值。

$$y = \begin{cases} 1, & \sum_i w_i x_i \geqslant \theta \\ 0, & \sum_i w_i x_i < \theta \end{cases} \tag{3}$$

在 19 世纪 50 年代末，Rosenblatt 第一次将神经元模型用于有监督的机器分类问题。他发明了感知机（perception）算法，能够具体调整神经元中连接的权重，

以达到学习的目的。随后，在分类问题是线性可分的前提下，感知机算法被证明可以收敛。

需要指出的是，早期神经元模型和现代感知机之间还是存在一定区别的。其中最主要的区别是它们采取的非线性激活函数不同。早期的神经元模型使用阈值判断阶梯函数作为非线性激活函数，而现代感知机常常使用诸如 Sigmoid、tanh 等连续的可导函数作为非线性激活函数。这主要是因为：用连续的可导函数作为非线性激活函数，使得从输入到输出的函数映射依然为连续可导函数，这对于优化时所需要的数学推导和计算往往更加便利。而当使用阈值判断阶梯函数作为激活函数时，从输入到输出的函数映射将变得不再可导。有趣的是，这种观点在近几年的深度学习中被重新审视。人们发现，在优化过程中，只要激活函数在大部分时候是光滑可导的即可，而不需要处处是光滑可导的。因此，人们不再局限于使用整体光滑可导的函数作为激活函数，诸如 ReLU 这种非整体可导的函数作为激活函数已被大量使用。

另外，需要指出的是，感知机在本质上只能够做线性分类，这一点从它的公式中就可以看出。因此，它只能够表达线性分界面，而无法表达更复杂的分界面。

1.2 现代神经网络

本节主要介绍现代神经网络发展中获得的一些重要成果。这些成果是一般神经网络所共有的，不局限于深度神经网络。具体来说，本节会介绍神经网络学习和优化的基石，即反向传播算法，以及著名的神经网络的通用函数近似性。

1.2.1 反向传播算法

反向传播是指将预测误差从输出层向输入层（即反向）进行传播，以便依次更新各层网络的参数（连接权重及偏置）。它通常结合某种参数更新的最优化方法（例如梯度下降法），一起实现参数的调整和更新。反向传播算法的推导基于

梯度计算的链式法则。反向传播算法的具体细节在任何一本神经网络的教科书中都可以找到，这里不再赘述。在此，我们仅讨论反向传播算法在深度神经网络学习中面临的重要困境和难题。

梯度消失问题是用反向传播算法训练深度神经网络时面临的最主要的难题之一。由于在反向传播算法中，每个神经网络的权重在每次训练迭代中进行更新，而其更新幅度与误差函数对当前权重的偏导数成正比。因此，当误差函数相对于当前权重的偏导数非常小的时候，那么当前权重的更新就会变得非常缓慢（更新量变得非常小）。在极端情况下，它甚至会使得神经网络变得完全无法更新，因而无法学习到合适的网络参数。

下面以常用的双曲正切函数作为激活函数来举例说明梯度消失问题。双曲正切函数的梯度取值范围是（0,1），当神经元的输入信号很大或者很小时，其输出值对于输入值的导数接近于 0，此时的神经元被称为处于“饱和”状态。这时，如果利用反向传播算法更新其输入信号对应边的权重，则更新量很小，更新过程很缓慢。这是因为，输出信号相对于输入信号的导数很小，而误差信号与输出信号仅差一个常量，因此，误差信号相对于输入信号的导数也很小。梯度消失问题对于网络层数比较多的深度神经网络尤为突出。在深层神经网络中，考虑到在反向传播中通过链式规则来计算梯度，那么误差信号对于前n层输入信号的梯度将是从后到前，将每一层神经网络输出信号与输入信号的梯度相乘。其中包含与n个取值范围在（0,1）的小数进行相乘，这意味着梯度（误差信号）随着n呈指数减小，这导致前层网络的训练会变得非常缓慢。

类似地，反向传播也可能遇到梯度爆炸问题。即在某种情况下，如果每一层的梯度取值恰巧都比较大（虽然双曲正切函数本身的梯度取值范围不会超过 1，但是神经元输入边的权重可能会大于 1，这使得输出信号对于输入信号的梯度仍可能大于 1），那么经过链式规则的多层相乘之后，梯度（误差信号）随着层数n呈指数增加，因此，当传递到前n层网络时，梯度值会非常大。这个问题被称为梯度

爆炸问题。然而，梯度爆炸问题相对梯度消失问题来说要简单得多。在实际中的做法往往是对梯度的幅度进行限制。如果根据链式规则计算得到的梯度值超过最大限度，则把它赋值为允许的最大值，然后再进行权重更新。

1.2.2 神经网络的通用函数近似性

我们可以将多个感知机并排放在一起，以得到一个单层神经网络；也可以将多个这种单层神经网络串联在一起，得到多层神经网络，我们称之为多层感知机。一个著名的结论是，多层感知机可以近似表示任何函数。对这个结论进行严格证明比较复杂。下面我们以一种非严格的解释形式来介绍其背后的原理。

让我们先从简单的情况讲起。解释含有一个隐藏的多层感知机可以近似表示任何布尔函数。这是因为每一个线性感知机都可以实现“与”逻辑或者“或”逻辑。举例来说，假设目标函数的输入是n维二值向量（即每一维取值为 0 或 1），目标函数的输出是各维输入经过“与”逻辑得到的输出。即只有各维取值都为 1 时才输出 1，否则输出 0。那么，为了近似表示这种布尔函数，我们可以构造一个感知机，即 sigmoid($k\cdot(x_1+x_2+\cdots+x_n-n+0.5)$)的感知机就可以近似实现其功能。其中，$k$的取值越大，近似程度越高。类似地，sigmoid($k\cdot(x_1+x_2+\cdots+x_n-0.5)$)这种形式的感知机可以近似表示“或”逻辑函数。进一步，由于每个布尔表达式都可以被转换成由多个“与”表达式进行 “或”组合所得到的联合表达式，因此，我们可以用上述两种感知机组合形成的感知机网络来模拟任何布尔函数。

下面讨论复杂一点的情况，即多层感知机可以近似表示任意有界连续函数。这是因为一个复杂的有界函数分界面，可以通过一系列线性分界面的组合并结合布尔函数来实现。具体来说，每一个感知机都对应着一个线性分界面，并可以将空间分成两部分。我们可以将分开的空间分别表示为 0 或 1。这样，经过n个线性分界面（对应n个感知机）的划分后，空间就被划分为一系列区域，而每一个有界区域都可以被表示为一个n维的布尔向量值，因此可以进一步将其转换为一

个布尔函数。由于每一个线性分界面都可以由一个感知机表达，而布尔函数也可由一个多层感知机表达，因此，任意有界连续函数均可由多层感知机来近似表示。

最后介绍最复杂的情况，即多层感知机可以近似表示任何函数。这是因为，根据上述介绍，多层感知机可以表示任意直径的圆（因为它是有界连续函数）。这样，我们可以添加另一个隐藏层用于执行“或”操作，以合并前述多层感知机输出的多个不同的圆，因此可用于近似表示任何函数。

1.2.3 深度的必要性

虽然神经网络在理论上可以近似表示任何函数，但当要表示的函数比较复杂时，所需的神经元个数往往会呈指数级增加，因此是不现实的。如何在保持其表达能力的同时减少所需神经元的个数，这个问题已经被研究了许多年。

此外，Bianchini 和 Scarselli（于 2014 年）将这项研究扩展到具有常见激活函数（包括 tanh 和 sigmoid）的神经网络。他们引入了贝蒂数（Betti number）的概念，并用这个贝蒂数来刻画神经网络的表达能力。他们的研究表明，对于浅层网络，网络的表示能力只能够随着神经元的个数进行多项式级别的增长；而在深层体系结构中，网络的表达能力可以随着神经元的个数呈指数级别的增长。

最近，Eldan 和 Shamir（于 2015 年）提出了一个更严格的证明，以标准的多层感知机网络及常用的激活函数为例，来说明网络的深度相比网络的宽度，能让描述能力呈指数级别的增长。他们的证明只依赖一些非常弱的假设，例如激活函数应该是一种温和递增函数、可度量等。

综上所述，我们有了较为扎实的理论基础来支持更深层的网络结构优于浅层的网络结构。然而，实际上，如果我们持续增加网络模型的深度，将会出现许多问题。其中，对于深层神经网络来说，网络模型参数学习难度的大幅增加是最突出的问题之一。

1.3 深度学习发展历史中的重要神经网络

下面介绍一下深度学习的兴起及其发展历程中的一些重要神经网络。这些重要的工作往往会给我们带来更深入的认识或新的思考角度，颇具启发性。

1.3.1 深度神经网络的兴起

虽然浅层（两个隐藏层）的多层感知机在理论上已经可以近似表示任何函数，但它需要的神经元个数往往随着目标函数的复杂程度而呈指数级增长。如何在保持其表达能力的同时减少所需神经元的数量，是在实际应用时所面临的一个关键问题。增加网络的深度，从而使底层网络的神经元节点被多次复用，以提高神经元的使用效率，被认为是解决这个问题的有效途径。因此，人们从对浅层神经网络的研究进一步转向对深层神经网络的研究。

从直观上讲，使用更深层的网络结构似乎是非常自然的事情。首先，人类大脑本身就是一个非常深层的网络结构；因此，从仿生学角度来看，深度神经网络更贴近人脑的构造。其次，人类对知识、概念等关键元素的组织往往呈现分层、分级的特点，即将最简单的事物抽象为最基本的概念，然后将低层次的概念进行组合得到较高层次的概念，而较高层次的概念经过组合得到更高层次的概念，依此类推。因此，人对概念的组织及思考模式，都是具有多层次结构的特点，而非很浅的一层或两层结构。事实上，现在的研究普遍认为，加深网络深度可以让所需的神经元个数呈指数级降低。具体命题表述如下：具有多项式个神经元的k层神经网络，如果使用$k-1$层结构的神经网络来表达，则需要用神经元个数以指数量级增才能具有相同的表达能力。需要指出的是，从理论上来讲，这个命题还没有被完全严格论证。因此，这是一个被普遍认同，但还没有完全严格证明的命题。

1.3.2 自组织特征映射

自组织特征映射（self-organizing festure map）通常被理解为是一种降维技术。自组织特征映射能使降维后的数据点（通常是两维）保留原空间数据点的

拓扑相似性。自组织特征映射也可以被看作是一种聚类技术，用于实现在聚类的结果上表达拓扑结构。

自组织特征映射是一种非监督学习网络，它使用一个邻近函数来保持输入空间的拓扑性质。与其他人工神经网络不同，自组织特征映射被应用于竞争学习而不是纠错学习。自组织特征映射的学习算法是使得网络模型的不同部分对某些输入模式进行类似的响应。具体的学习细节可参考相关论文，这里不再赘述。[1]

1.3.3 霍普菲尔德神经网络

霍普菲尔德神经网络（Hopfield neural network）是一个完全连接的神经网络，具有二进制阈值神经单元。这些神经单元的输出值为 0 或 1。这些神经单元通过双向权重进行全连接。在这种设置下，霍普菲尔德网络的能量定义为下式：

$$E = -\sum_i s_i b_i - \sum_{i,j} s_i s_j w_{i,j} \tag{4}$$

其中，s是神经元的状态（即神经元的输出值），b表示偏差，w表示联接的权重，下标i和j表示神经元的索引。这个能量函数是由旋转玻璃理论中的势能函数演化而来的。

【知识点讲解】旋转玻璃理论是用于描述磁现象的物理学理论。它的直观解释如下：

当一组偶极子被放置在一起时，每个偶极子都被迫对齐所有这些偶极子在其位置共同产生的场。但是，当某个偶极子（例如偶极子a）调整自身朝向以便对齐它所在位置的场时，它的改变影响了其他位置的场，导致其他偶极子改变朝向。而其他偶极子朝向的改变又进一步导致偶极子a所在位置的场的改变，进而使得偶极子a进一步调整朝向，以对齐它所在位置的场。这种变化会持续进行并最终

1 KOHONEN T. Self-Organized Formation of Topologically Correct Feature Maps. Biological Cybernetics[J] 43 (1), 1982:59–69.

收敛到一个稳定状态，此时，整个磁系统将具有最小的势能。

当将某一个输入状态呈现给网络时，在给定的网络权重下，霍普菲尔德神经网络可以搜索得到某个隐藏状态，使得（4）式所示的能量函数最小化，而这个最小化能量函数就可以作为当前输入状态的最小能量函数。霍普菲尔德神经网络学习的目的就是寻找其网络权重的一组解，使得此网络在给定的一组输入状态下，各输入状态对应的最小能量函数的和最小。

霍普菲尔德神经网络通常可以被用于编码或记忆数据状态。它的一个明显缺点是，它无法高效地实现数据编码或记忆。由于这个缺点，霍普菲尔德神经网络逐渐淡出深度学习的热点领域，被其他受其启发的模型所取代。

1.3.4 玻尔兹曼机及受限玻尔兹曼机

玻尔兹曼机是受霍普菲尔德神经网络模型的启发，对其进行改进升级而发明的新算法。玻尔兹曼机是由玻尔兹曼分布而得名的。玻尔兹曼分布最初用于描述系统在各种可能状态下的粒子概率分布，如下式所示：

$$F(\mathrm{s}) \propto \mathrm{e}^{-\frac{E_s}{kT}} \tag{5}$$

其中，s代表系统的状态，E_s是系统在状态 s 下对应的能量。k是玻尔兹曼常数。相应地，系统处在状态s的概率可以由下式得到：

$$p_{s_i} = \frac{\mathrm{e}^{-\frac{E_{s_i}}{kT}}}{\sum_j \mathrm{e}^{-\frac{E_{s_j}}{kT}}} \tag{6}$$

玻尔兹曼机是一个具有隐藏节点的随机霍普菲尔德神经网络。现在最常用的是它的一个变种，即受限玻尔兹曼机（restricted Boltzmann machine）。受限玻尔兹曼机是由m个可见节点和n个隐藏节点构成的网络模型，如图 1.2 所示。其中，可见节点之间互相没有边连接，隐藏节点之间也没有边连接；每个可见节点只和n个隐藏节点有边连接；而每个隐藏节点也只和m个可见节点有边连接。

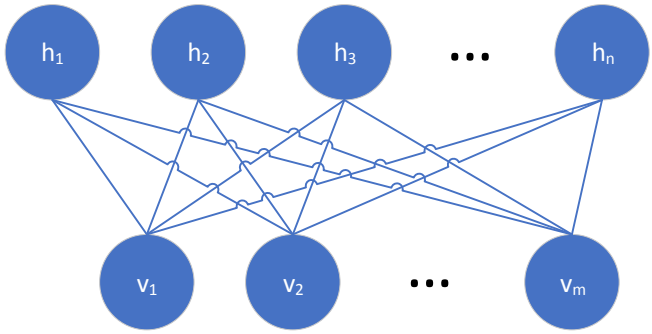


图 1.2　受限玻尔兹曼机

在受限玻尔兹曼机的能量函数定义中，除可见节点之间无连接，隐藏节点之间无连接外，其他形式与霍普菲尔德神经网络的能量函数定义相同，如下式所示。

$$E(v,h) = -\sum_{i=1}^{n}\sum_{j=1}^{m} w_{ij}h_i v_j - \sum_{j=1}^{m} b_j v_j - \sum_{i=1}^{n} c_i h_i \tag{7}$$

其中，v表示可见节点，h表示隐藏节点。

基于上述能量函数，可见节点和隐藏节点状态的联合概率分布可以定义为下式：

$$p(v,h) = \frac{e^{-E(v,h)}}{\sum_{v',h'} e^{-E(v',h')}} \tag{8}$$

其中，v和h表示可见节点和隐藏节点的某一个具体状态；v'和h'表示可见节点和隐藏节点可能出现的任意一种状态。

根据上式联合概率分布的表达式，我们可以进一步得到可见节点或者隐藏节点的边缘概率分布，这些边缘分布可以通过对联合概率进行求和得到。例如，通过对隐藏节点所有可能的状态进行求和，可以得到可见节点的概率分布，如下式所示。而这个概率分布可以用来对可见节点的状态进行采样，即产生数据。

$$p(v) = \frac{\sum_h e^{-E(v,h)}}{\sum_{v',h'} e^{-E(v',h')}} \tag{9}$$

训练受限玻尔兹曼机的常用方法是最大似然法，即确定一组参数，使得观察

数据（即训练样本）在这组参数下根据模型计算得到的概率值最大。我们通常是将似然函数取对数，将其作为优化目标函数，并根据此目标函数进行梯度下降以求解最优参数。但是，由于此目标函数中的分母项需要针对一个极大的集合（系统所有可能的状态）进行求和，因此，对受限玻尔兹曼机这个模型来说，严格求解其梯度在实际中几乎是不可能的。随后，有研究人员提出利用马尔可夫链蒙特卡洛（Markov Chain Monte Carlo，MCMC）采样来实现快速近似求解。凭借合理的建模能力及快速的逼近求解算法，受限玻尔兹曼机很快引起了极大的关注，并且成为深度神经网络发展中的重要基石之一。

1.3.5 深度信念网

将多个受限玻尔兹曼机堆叠在一起，就构成了深度信念网（deep belief net）。需要指出的是，与简单地将多个受限玻尔兹曼机堆叠在一起不同的是，深度信念网只允许在顶层有双向连接（注：可见层被看作是底层），而其他层均是只有自顶向下的单向连接。我们可以将深度信念网理解为多层生成模型；或者将它视为一种具有扩展层的单层受限玻尔兹曼机，其中，扩展层专门用于生成数据模式。

在深度信念网的参数学习中，它首先通过贪婪算法的方式进行逐层训练，作为预训练步骤；然后以预训练得到的网络参数作为初值，在整个网络上应用标准的反向传播算法并根据训练数据对网络的参数进行进一步学习，实现整体的精细调整以进一步优化网络。引入逐层的预训练是成功训练深度信念网的关键因素。逐层预训练首先按层自下而上地训练网络模型：将前两层作为受限玻尔兹曼机处理并训练它，然后将第二层和第三层作为另一个受限玻尔兹曼机处理并训练它，依此类推。实践证明，这种预训练对后续的整体精细调整至关重要。只有在预训练的基础之上，后续的精细调整才能够取得好的效果，深度信念网的参数才能得到合适的优化解。逐层预训练的想法很简单，但它起作用的根本原因仍然被广泛研究。现在，一个普遍认同的看法是，无监督的预训练将模型初始化到参数空间

中的一个优化点，这个优化点能导致优化过程的效率更高，以至于在优化过程中可以找到使代价函数更小的解。此外，还有人从正则化的角度解释预训练机理。这一类观点认为，无监督预训练指导学习过程趋向于找到代价函数取值很小的盆地区域，这个初始化实际上限制了利用梯度下降求解所能达到的最终解空间的范围，在一定程度上实现正则化，使得模型可以在训练数据集上进行更好的推广。除深度信念网外，逐层预训练机制也被广泛应用于很多其他模型中，例如自编码器及其各种变种。

1.3.6 其他深度神经网络

除上述介绍的网络外，还有很多深度神经网络，例如自编码器（Auto-Encoder）、卷积神经网络（Convolutional Neural Net，CNN）、递归神经网络（Recurrent Neural Network，RNN）、生成对抗网络（Generative Adversarial Network，GAN）等。在本书的后续章节中会专门介绍卷积神经网络、递归神经网络，以及生成对抗网络，因为这几类神经网络在自然语言处理及图像处理领域有着非常广泛的应用。

1.4 本章小结

在本章中首先介绍了神经网络发展的历史，以及神经网络发展初期一些比较著名的工作。那时的神经网络仍是非常简单的“浅”网络。然后介绍了神经网络的通用函数近似性，并论述了神经网络从“浅”到“深”的必要性，以及经典的反向传播算法在网络层数变深时所面临的一些主要困难。在此之后介绍了深度神经网络在发展初期的一些重要工作及其核心思想。本章概述的这些神经网络，大多已经被更先进的网络所取代。但它们所蕴含的核心思想，仍然值得大家推敲和体会。在当前的主流应用中，卷积神经网络、递归神经网络和生成对抗网络等是应用最为广泛的现代神经网络。在后续章节中会陆续进行具体介绍。

参考文献

[1] KAWAGUCHI K. A multithreaded software model for backpropagation neural network applications. ETD Collection for University of Texas, El Paso. AAIEP05411,2000.

第 2 章

深度学习开源框架

想要学习深度学习，就要掌握一个有效的深度学习框架。那么现在都有哪些主流的深度学习开源框架，它们各自的优/缺点又是什么呢？

在工程实践中，我们认为，要选择一个得心应手的深度学习框架，主要从框架的流行程度和应用场景出发。对想要入门深度学习的用户来说，选择一个比较流行的框架意味着能从多种渠道中获得更多的支持，譬如书籍、博客、GitHub项目等；对专门从事深度学习模型开发的用户来说，选择一个适合应用场景的框架，使用一门合适的开发语言，将会使模型开发达到事半功倍的效果。

下面介绍现在主流的深度学习开源框架，帮助读者对这些框架有直观的认识。

2.1 主流的深度学习开源框架

这是一个深度学习框架群雄逐鹿的年代，许多深度学习框架在涌现、合并，抑或凋零，其中包括 TensorFlow、Keras、CNTK、PyTorch、MXNet、Theano等，它们的背后是 Google、Microsoft、Facebook、Amazon 等科技巨头。目前来看，在 Google 的号召下，TensorFlow 拥有众多的开发者，在关注度和用户数上都占据绝对优势。Keras 作为“框架上的框架”，以其易用性吸引了不少支持者。Facebook 的 PyTorch、Amazon 的 MXNet、Microsoft 的 CNTK 等各有一技之长，在目前深度学习的战场上立有一席之地。而曾经的霸主 Theano，在没有大公司的支持下，开始没落。

对准备入门深度学习的人来说，深度学习框架在 GitHub 上的活跃度可以作为一个重要的参考指标。在 GitHub 上，被 Star（收藏）和 Fork（分支）越多，意味着有越多的开发者选择了这个深度学习框架。我们总结了在 GitHub 上（到 2019 年 2 月 1 日为止）各个主流深度学习框架的活跃度，以及它们对编程语言和系统的支持情况，如表 2.1 所示。

表 2.1　主流深度学习框架的活跃度

框架名称	GitHub 的活跃度			维护团队	支持语言	支持系统	编程模式
	Star（数量）	Fork（数量）	贡献者数量				
TensorFlow	119730	71727	1817	Google	Python、C++、Java、Go	Linux、macOS、Windows、Raspberry Pi	符号式编程
Keras	37850	14450	771	Google	Python、R	Linux、macOS、Windows	符号式编程
PyTorch	24331	5778	906	Facebook	Python、C++	Linux、macOS、Windows	命令式编程
MXNet	16255	5842	667	DMLC、Amazon	Python、C++、Clojure、Julia、Perl、R、Scala、Java	Linux、macOS、Windows、Raspberry Pi、NVIDIA Jetson	符号式编程和命令式编程的混合编程
CNTK	15721	4213	191	Microsoft	Python、C++、C#/.NET、Java	Linux、Windows	符号式编程
Theano	8667	2479	334	蒙特利尔大学	Python	Linux、macOS、Windows	符号式编程

从深度学习开源框架统计表中，我们可以看到：

- TensorFlow 处于毫无疑问的霸主地位，其 Star 数和 Fork 数均超过了表中其他开源框架的 Star 数和 Fork 数的总和。
- 在对语言的支持上，Python 为最受欢迎的语言，所有框架均支持 Python 或者 Python 优先。Python 的优雅、简洁，使得其具有较低的使用门槛和极高的编程效率，能够更快地尝试不同的参数组合以及更复杂的模型。C++排名第二，除 Theano 和 Keras 外，其他深度学习框架底层都是由 C/C++语言编写的，保证了模型训练的速度和效率。其他语言支持的框架各有不同，譬如 TensorFlow 支持 Go 语言、MXNet 支持 Scala 语言、CNTK 支持 C#语言等。
- 在支持的系统上，所有框架基本都支持 Linux、macOS 和 Windows 这 3 个主流系统（除 CNTK 缺少对 macOS 的原生支持外），TensorFlow 和 MXNet 还支持 Raspberry Pi（树莓派），MXNet 还支持 NVIDIA Jetson。

- 在编程模式上，Theano、TensorFlow、CNTK、Keras 为符号式编程；PyTorch 为命令式编程；MXNet 比较特殊，支持符号式编程和命令式编程的混合编程。

接下来会逐个讲解各个深度学习框架的特点和背景。

首先介绍一下深度学习框架的两种不同的编程模式，它们对代码的编写、调试及运行速度都有不同的影响。

【知识点讲解】：编程模式与计算图

编程模式通常分为命令式编程（imperative style programs）和符号式编程（symbolic style programs）。

符号式编程将计算过程抽象为计算图。使用计算图可以方便地描述计算过程，所有输入节点、运算节点、输出节点均被符号化处理。计算图通过建立输入节点到输出节点的传递闭包，从输入节点出发，沿着传递闭包完成数值计算和数据流动，直到输出节点。这个过程经过计算图优化，以数据流方式完成，节省了内存空间，计算速度快，但不适合程序调试。因为符号式编程中的计算图先定义后执行（define and run），也被称为静态计算图。

命令式编程就是输入什么便执行什么，在运行语句时马上进行计算，对语句基本没有优化，按原有逻辑执行，容易理解和调试。命令式编程在运行过程中定义（define by run）的计算图，是动态计算图。

Theano

从表 2.1 中可以看出，Theano 的活跃度排名倒数第一，但是在 2015 年，它还在深度学习框架中处于“霸主”的地位。Theano 诞生于 2008 年，由蒙特利尔大学 Lisa Lab 团队开发并维护。[1]它是第一个被大规模使用的深度学习框架，

1 BERGSTRA J, BREULEUX O, BASTIEN F, et al, Theano: A CPU and GPU math compiler in Python. Proceedings of the 9th Python in Science Conf 2010 [C], Vol. 1, 3-10.

也是其他深度学习框架的基石。Theano 是一个完全基于 Python 的符号计算库，采用符号式编程，专门为处理大规模神经网络训练的计算而设计。

在还没有 Theano 时，研究者若想用 Python 来实现深度学习算法，则颇为麻烦，仅能用 Python 的 NumPy/Scipy，还需要手动推导各类函数并用代码实现，成本很高。大家迫切需要一个通用的深度学习库。Theano 就是这样发展起来的，它能和数值计算无缝对接，注重性能和稳定性，惰性求值，且可以自动在 GPU 上运行等，这节约了研究者不少成本，大大提高了研究效率。有许多开发人员为它编写了高质量的文档和教程，用户可以方便地查找 Theano 的各种问题的答案。并且，在 Theano 之上派生出了大量基于它的深度学习框架和上层封装，譬如 Keras、Lasagne、pylearn2 和 Scikit-theano 等，可谓是一个庞大的家族。可以说，如果没有 Theano，则可能根本不会出现这么多好用的深度学习框架，或者，至少会让目前深度学习框架百花齐放的局面晚出现许多年。

但是，随着时间的推移，许多新的深度学习框架的崛起，Theano 的劣势也开始显露出来：Theano 在底层执行上相比其他深度学习框架效率低，这使得其只能作为研究工具，很难作为产品来使用。其在编译时需要将用户的 Python 代码转换成 CUDA 代码，再编译为二进制可执行文件，并且对复杂模型的编译时间长。同时，Theano 还没有分布式的实现，在 CUDA 和 cuDNN 上不支持多 GPU。在各大公司纷纷推出或扶植深度学习框架之后，缺少持续的支持和维护，对 Theano 来说是比较致命的，Theano 的迭代更新相比其他框架要慢许多。

令人遗憾的是，随着深度学习框架的竞争越来越激烈，2017 年 9 月 29 日，在发布了最新版 Theano 1.0 之后，Theano 和大家告别了。其核心贡献者，Yoshua Bengio 和 Pascal Lamblin 写道："我们将会继续做微小的维护让 Theano 继续工作一年，但是我们将会停止开发新的特性。"关于 Theano 离开的原因，Yoshua Bengio 的解释是，一方面，出现了许多新的深度学习框架并快速进化，它们继承了 Theano 的创新，弥补了 Theano 的不足；另一方面，比较陈旧的代码库则

是 Theano 创新的一大阻碍。

TensorFlow

TensorFlow 由 Google 于 2015 年 11 月发布，是一个比较成熟和完善的深度学习框架。得益于 Jeff Dean 等人及 Google 的强大号召力，TensorFlow 一经推出就得到了迅猛发展，其在 2015 年 11 月刚开源的第一个月就积累了 10000 多的 Star 数。受“先驱”Theano 的影响，TensorFlow 也采用符号式编程。

TensorFlow 有两个重要的概念：Tensor（张量）和 Flow（流）。在几何代数中定义的张量是基于向量和矩阵的推广。在编程时，我们可以把一个张量想象成一个 n 维的数组或列表。Flow 被直译为“流”，其直观地表达张量之间通过计算来相互转换的过程。深度学习的一系列任务：分类、聚类、回归等，其实都是在给定输入和输出下，探索其内部函数映射的过程。张量与流通俗易懂地解释了深度学习算法的运行过程，即张量之间通过计算相互转换。TensorFlow 的运行机理为：用张量定义数据模型，把数据模型和操作定义在计算图中，使用会话运行计算。

TensorFlow 对深度学习的支持广泛，它通过 SWIG（Simplified Wrapper and Interface Generator）实现对 Python、C++、Java、Go 等多种编程语言的支持；同时支持所有主流的深度学习算法。TensorFlow 可以在主流的操作系统和移动平台中运行，并适用于多个 CPU、GPU 或 TPU[1]组成的分布式系统。TensorFlow 提供可视化组件 TensorBoard，能可视化网络结构和训练过程。同时，TensorFlow 提供 TensorFlow Serving，这是一个为生产环境而设计的机器学习服务系统，可以很方便地导出和部署 TensorFlow 模型。TensorFlow 得到了许多云平台的支持，包括亚马逊的 AWS、微软的 Azure、Google Cloud、阿里云、腾讯云等，开发人员能够更为便捷地使用 TensorFlow，而不需要自己来

1 TPU（Tensor Processing Unit，张量处理单元）是一款 Google 于 2016 年 5 月推出的处理器，专门为深度学习使用，速度上比 CPU 和 GPU 快数十倍。

搭相关环境。有了这些组件的支持，TensorFlow 实现了从部署环境、训练模型、调试参数，到打包模型，最后部署服务的全流程。

TensorFlow 一开始被饱受诟病的是其对分布式的支持，在 2015 年刚发布时，TensorFlow 只支持单机。那时，每当框架之间进行性能对比评测时，TensorFlow 经常会被作为较差的对照组。但是，凭借 Google 的支持，TensorFlow 发展迅速，在单 GPU 上的性能追上了其他框架；TensorFlow 对分布式的支持很好，支持 Parameter Server-Worker 和 AllReduce 的分布式技术。目前来说，TensorFlow 的缺点是比较庞杂，它的版本更新快，有一些常用的函数经常被挪动位置，用户在写代码时常常需要根据目前版本翻阅文档。同时，TensorFlow 追求对机器学习支持的大而全，其封装比较混乱，这对初学者来说入门成本略高；其另一个薄弱的地方在于静态的计算图，使得一些计算的实现颇为复杂，譬如动态 RNN、Seq2Seq 中的 Beam Search 算法。

总的来说，瑕不掩瑜，TensorFlow 对多平台、多语言支持的特性，相比其他深度学习框架具有先发优势。其背后又有 Google 强大的开发、维护能力的支持，以及对深度学习生态全方位的支持，这些使得其成为目前最为流行的深度学习框架。

【知识点讲解】分布式架构

对深度学习框架来说，对分布式的支持是很重要的内容，因为在深度学习领域，单机已经处理不了海量的数据和参数了。前文中提到的 Theano 被淘汰的一个很重要的原因就是不支持分布式架构。在深度学习框架中，主要有两种不同的分布式架构，分别是 Parameter Server-Worker 和 AllReduce。

在 Parameter Server-Worker 架构中，有两种不同职责的执行单元（我们不妨将其理解成机器），分别是 Parameter Server（参数服务器）和 Worker。参数服务器负责存储参数，Worker 负责处理数据和任务。在有多个参数服务器和多个 Worker 时，每个参数服务器实际上只负责分到的部分参数，每个 Worker

也只负责分到的部分数据和任务。Parameter Server-Worker 的架构如图 2.1 所示，所有的参数服务器一起维持着一个全局的共享参数，从 Worker 处收到参数的梯度信息$\nabla f_i(x)$后更新参数：$w_{t+1} = \text{UPDATE}\big(w_t, \eta, \nabla f_i(x)\big)$；Worker 在每次处理数据前都会从参数服务器中拿到最新的参数w_{t+1}，然后针对当前数据计算新的梯度。

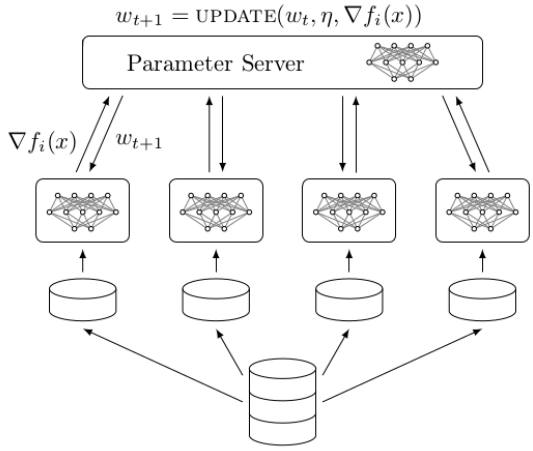


图 2.1　Parameter Server-Worker 分布式架构

AllReduce 是另外一种分布式架构，其没有单独的参数服务器。每台机器既负责一部分计算，也保存并更新模型参数。假设有 n 台机器执行 AllReduce，其计算过程是将数据分成 n 份，分给每台机器分别执行，对所有机器的结果汇总即可计算平均梯度，然后将平均梯度发往各台机器中以更新参数。图 2.2 展示了一个有 4 台机器、4 个参数的 AllReduce 的执行过程。在开始时，4 台机器分别分到 1/4 的数据，各自计算 4 个参数的梯度。待 4 台机器计算完成后，通过 AllReduce 计算这些参数的平均梯度，然后将平均梯度发给各个机器更新其本地的参数。其中，百度提出的 Ring-AllReduce 算法能够快速进行 AllReduce 操作，如图 2.3 所示。在这种实现中，每次机器计算梯度结束，每台机器对应的参数就会沿着圆环向前传递，依次合并每台机器的计算结果，参数绕圆环一周即完成了更新。

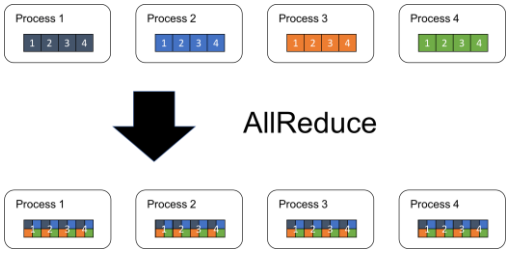


图 2.2 AllReduce 算法图解

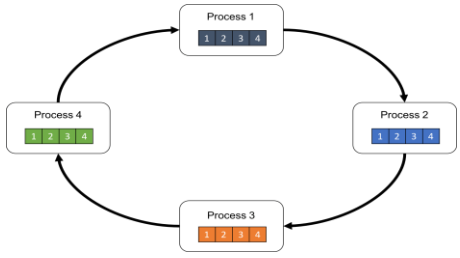


图 2.3 Ring-AllReduce 算法图解

PyTorch

PyTorch 是由 Facebook 于 2016 年 10 月推出的深度学习框架。[1] PyTorch 经历了多次变迁，其前身是 Torch（2002 年 10 月推出）；2018 年 4 月 1 日，PyTorch 又合并了 Facebook 大力支持的另一个框架 Caffe2。2018 年 10 月 1 日，Facebook 正式发布了 PyTorch 1.0 预览版。对于 PyTorch 与 Caffe2 的合并，Caffe2 的作者贾扬清表示："PyTorch 有优秀的前端，Caffe2 有优秀的后端，将其整合以后可以进一步最大化开发者的效率。"

PyTorch 是一个 Python 优先的深度学习框架，其目标是让设计科学计算算法变得更便捷。PyTorch 主要吸引两类人使用：一类是代替 Python 的 NumPy 包来更好地使用 GPU 性能的人；另一类是想体验 PyTorch 这个灵活和高速的深度学习研究平台的人。在编程语言上，除 Python 外，PyTorch 也支持使用 C/C++ 语言编写神经网络模型。PyTorch 可以在 Windows、Linux、macOS 操作系统上运行，并且适用于多个 CPU/GPU 组成的分布式系统。在云平台方面，同 TensorFlow 类似，PyTorch 也得到了包括亚马逊的 AWS、微软的 Azure、Google Cloud、阿里云、腾讯云等云平台的支持。

对于 TensorFlow，PyTorch 一直是追逐者的姿态，PyTorch 相对比较新，

1 PASZKE A, GROSS S, CHINTALA S, et al, PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration[EB], 5(2017):6.

其社区规模较小，不过其文档更为规整，对学习者更为友好。PyTorch 也提供了可视化工具 tensorboardX，并支持画动态图。对比静态的 TensorFlow，PyTorch 的动态神经网络能更高效地处理一些问题，譬如对 RNN 这种需要变化时间长度的网络结构。同时，PyTorch 一直宣传其具有效率和速度的优势，以及 Python 优先的易用性，在用户开发新的深度学习模型时，值得尝试一下 PyTorch。PyTorch 在 GitHub 上的 Star 数和 Fork 数迅猛增加，这也证明了用户对其前景的看好。

MXNet

MXNet 最初出现在 2015 年 12 月 NIPS 的机器学习系统 Workshop[1]中，是由 DMLC（Distributed Machine Learning Community，分布式机器学习社区）开发的深度学习库。2016 年 11 月 22 日，亚马逊 CTO Werner Vogels 在其博文中写道：MXNet 被 AWS 正式选为其云计算的官方深度学习平台。”2017 年 10 月，亚马逊和微软为 MXNet 推出了一个名为 Gluon 的深度学习新接口，使构建深度学习模型更为容易。

MXNet 是一个全功能、灵活、可编程和具有高扩展性的深度学习框架。它是在众多框架中率先支持多 GPU 和分布式的，同时其分布式性能也非常高。MXNet 支持符号式编程和命令式编程的混合模式，以最大化执行效率和提高产出。MXNet-Gluon 的 HybridBlock 和 hybridize 接口可以在静态和动态之间实现一键切换，让用户在享受动态编程灵活易用的同时最小化性能的损失。在云平台方面，MXNet 率先得到了亚马逊云平台 AWS 的支持，现在，开发人员也能够在微软的 Azure、Google Cloud、阿里云、腾讯云等云平台使用。

同 TensorFlow 一样，MXNet 通过 SWIG 实现了对 C++、JavaScript、Python、R、Perl 等多种编程语言的支持。开发者可以选择自己熟悉的编程语言

1 Chen T Q, Li M, Li Y T, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems[C]. arXiv preprint arXiv:1512.01274, 2015.

进行开发工作。在 MXNet 的后端，所有代码都以 C++编译，因此，无论构建模型使用的是哪种编程语言，都能实现高性能开发。

MXNet 比较令人诟病的是其文档和教程：API 文档略差，自定义教程的缺乏使得用户上手比较困难。MXNet 的社区略少，有时用户遇到问题还需要自己去查阅修改源码，导致使用门槛略高。MXNet 也在积极改进，其文档正在逐步完善，社区也在逐步壮大。

CNTK

认知工具集（Cognitive Toolkit，CNTK），是由微软于 2016 年 1 月 25 日推出的深度学习框架。[1] CNTK 起源于 2014 年微软研究院的黄学东博士和他的团队设计的一套内部工具，其用来帮助更快改进计算机理解语音的能力。这套内部工具成为目前广为人知的认知工具集（CNTK）的基础，也是微软认知工具集名称的来源。

CNTK 最开始用于语音识别领域，目前已经发展成一个通用的、跨平台的深度学习框架。与其他深度学习框架相比，微软一直强调 CNTK 具有速度优势。CNTK 宣称比 TensorFlow 快很多，特别是在 RNN 上，可以快 5~10 倍。CNTK 为随机梯度下降（Stochastic Gradient Descent，SGD）提供了独特的 1 比特 SGD，其对每个次梯度（subgradient）进行量化，每个值压缩到 1 比特。这项技术在保持准确度不变的情况下将性能提升了 10 倍。在云平台方面，CNTK 得到了微软自家云平台 Azure 的支持，现在也能在亚马逊的 AWS、Google Cloud、阿里云、腾讯云等云平台中使用。

在编程语言上，CNTK 支持 Python、C#和 C/C++语言；在操作系统上，CNTK 可以在 Windows、Linux 系统上运行，并适用于多个 CPU/GPU 组成的

1 SEIDE F, AGARWAL A. CNTK: Microsoft's open-source deep-learning toolkit. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 2016 [C], ACM, 2135-2135.

分布式系统。同时，到目前为止，CNTK 对 ONNX（一个支持不同深度学习框架生成的模型相互转换的开源项目）的支持最好。CNTK 对 C#/.Net 的支持也是其一个亮点。与 MXNet 类似的是，CNTK 的劣势是其影响力相对有限，核心贡献开发者偏少，文档略差，用户上手略复杂。

2018 年 11 月，微软对 CNTK 的态度也有所转向，其宣布继续支持 CNTK 的更新，但更多工作在 Facebook 的 PyTorch 的支持上。微软的 Azure 也已经支持 PyTorch 和 TensorFlow。CNTK 的命运也将如 Theano 那般，不禁让人唏嘘。

Keras

Keras 出现于 2015 年，由 François Chollet（现为 Google 的人工智能研究员）创立，其目标是提供一个简洁、易用的深度学习框架。与之前介绍的几个深度学习框架不同的是，Keras 并不是一个独立的深度学习框架。Keras 最初是构建于 Theano 之上的封装框架，使用 Keras 需要安装 Theano，但是其自包含（self-contained）的设计，使得其在随后的发展中能够选取不同的后端引擎，包括 TensorFlow 和 CNTK，Theano 的结束并不意味着 Keras 的终点。Keras 的简洁、易用，使其收获了不少粉丝。目前 MXNet 也在积极开发，以支持 Keras。

Keras 是一个基于 Python 的深度学习库，其旨在帮助用户进行快速的原型实验，以最小的时延把想法转换为实验结果。使用 Keras 搭建网络和训练网络非常容易。而且，在通常情况下，如果需要深入模型中控制细节，使用 Keras 提供的一些函数就可以了，很少需要深入其后端引擎中。Keras 发展迅速，其特性主要包含：符号式编程、支持 Python、快速生成原型、高度模块化、易扩展。Keras 的模块简洁、易懂、可配置，可以自由组合神经网络、损失函数、优化器、初始化方法、激活函数和正则化等模块。模块化简化了编写深度学习模型的复杂度，缩短了尝试新网络结构的时间。因此，Keras 适合于快速生成原型。Keras 的模型定义可以直接由 Python 脚本完成，而不需要额外的配置文件来定义，更方便用户调试模型和超参数。若将 TensorFlow 看作 C++，重而全面；Keras 就如同

Python，简洁优雅。因此，Keras 社区发展得很快，其目前在 GitHub 上的 Star 数和 Fork 数也很多。

因为 Keras 具有简洁、模块化的特性，从目前来看，Keras 是人工智能领域中对新手最友好，也是最易于使用的深度学习框架之一。

闲话 ONNX

在 Google 的 TensorFlow 发展壮大，在开源社区中获得了最多的开发者之时，出现了一个拥有多家公司支持的开放式神经网络交换项目。

2017 年 9 月，微软和 Facebook 发布了开放式神经网络交换（Open Neural Network Exchange，ONNX）项目，希望打造一个开放的深度学习开发生态系统。其目的是帮助 AI 开发者能够随着项目的发展而选择正确的框架。ONNX 可以实现模型在主流深度学习框架之间的切换，譬如使用一个框架进行训练，之后转移到另一个框架进行推断。在项目的不同阶段，AI 开发者可以选择适合其项目当前阶段的框架，并随着项目的发展在不同框架之间进行转换，这帮助 AI 开发者缩短了从研究到生产的路径。

之后在 2017 年 10 月，AMD、ARM、华为、IBM、英特尔、高通等公司也宣布支持 ONNX。2017 年 11 月，亚马逊的 MXNet 加入 ONNX 开放生态，提供 ONNX-MXNet 开源 Python 包，帮助 ONNX 将深度学习模型导入 MXNet 中。

到本书付梓之时，Google 的 TensorFlow 并未支持 ONNX 项目。深度学习框架目前的格局，颇有一些三国时“联吴抗魏”的味道。感兴趣的读者可以访问 ONNX 项目官网了解更多内容。

2.2 简单神经网络模型在不同框架上的实现对比

了解了深度学习的框架之后，下面用不同的深度学习框架来完成一个手写体识别任务——一个在深度学习领域类似于实现“Hello World”的任务。通过这个任务，我们可以直观地感受到不同深度学习框架实现的代码差异。

我们采用一个开放的手写体识别数据集：MNIST 数据集，它在许多论文、教程中被广泛使用。[1]MNIST 数据集来自美国国家标准与技术研究所。数据集由 250 个不同的人手写的数字构成，其中 50%来自高中学生，50%来自人口普查局的工作人员。它包含了 4 个部分：

（1）训练图片集（training set images）：train-images-idx3-ubyte.gz（9.9 MB，解压后 47 MB，包含 60,000 个样本）。

（2）训练标签集（training set labels）：train-labels-idx1-ubyte.gz（29 KB，解压后 60 KB，包含 60,000 个标签）。

（3）测试图片集（test set images）：t10k-images-idx3-ubyte.gz（1.6 MB，解压后 7.8 MB，包含 10,000 个样本）。

（4）测试标签集（test set labels）：t10k-labels-idx1-ubyte.gz（5KB，解压后 10 KB，包含 10,000 个标签）。

其样例如图 2.4 所示。

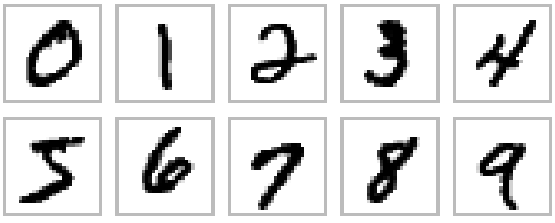


图 2.4　MNIST 数据集样例

对于这个数据集，我们会统一构造一个三层的神经网络，分别是输入层、隐藏层和输出层。这是一个简单的前馈神经网络，通常也被称为多层感知机（Mult-Layer Perceptron，MLP）。

其结构如图 2.5 所示。

1 这份数据集可在 http://yann.lecun.com/exdb/mnist/中获取。

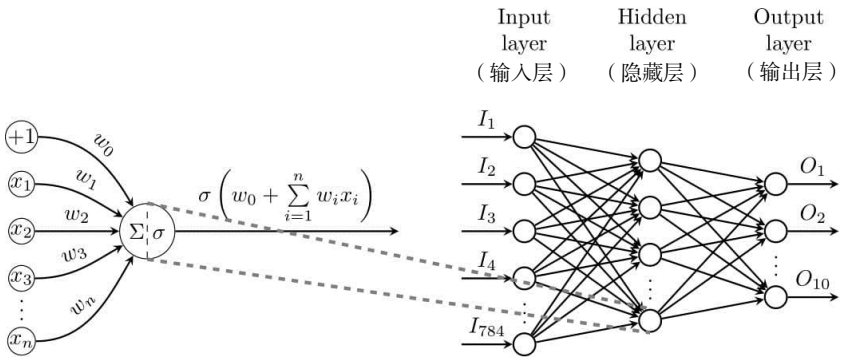


图 2.5　简单神经网络结构

该神经网络有三层，第一层（输入层）有 784 个神经元（每张手写体识别图像均为 28 像素 ×28 像素的图像），第二层（隐藏层）有 200 个神经元，最终层（输出层）有 10 个神经元（有数字 0~9 共 10 个类别）。我们使用 Sigmoid 函数作为激活函数，将均方误差作为损失函数，使用 Adam 优化器，学习率为 0.01。

所有实现都遵循相同的步骤：

（1）设置参数并加载数据集（大多数框架都有加载标准数据集的方法，如 MNIST）。

（2）创建多层感知机（MLP）神经网络结构。

（3）定义训练函数，包括模型训练和模型存储。

（4）定义预测函数，包括模型导入和测试数据预测。

（5）创建 main 函数，让用户使用训练数据集进行训练，然后使用测试数据集进行预测。

在本章中，我们使用的深度学习框架的版本分别是：TensorFlow 1.12.0、Keras 2.2.4、PyTorch 1.0、MXNet 1.3.1、CNTK 2.6。

在以下代码中，我们将其整理成了相同的格式进行比较，每份代码包含 5 个部分：设置参数并加载数据集、定义神经网络、定义训练函数、定义预测函数和 main 函数。其中，main 函数在每个代码中均相同，为：

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--action", type=str, default="predict" )
    FLAGS, unparsed = parser.parse_known_args()
    if FLAGS.action == "train":
        train()
    if FLAGS.action == "predict":
        predict()
```

运行代码的方法：

（1）训练：python [脚本名].py --action train

（2）预测：python [脚本名].py --action predict

TensorFlow

- 设置参数并加载数据集

```
import tensorflow as tf
import argparse
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data

# 基本参数
inputs, hiddens, outputs = 784, 200, 10
learning_rate = 0.01
epochs = 50
batch_size = 64

# 导入数据集
mnist = input_data.read_data_sets("./mnist/", one_hot=True)
x = tf.placeholder(tf.float32, [None, inputs])
y = tf.placeholder(tf.float32, [None, outputs])
```

在这里，我们定义了基本的参数，包括输入层维度（inputs），隐藏层维度（hiddens）、输出层维度（outputs）、学习率（learning_rate）、迭代次数（epochs）、数据块大小（batch_size）。同时，将数据输出设置为 one_hot 编码。

- 定义神经网络

```
# 神经网络结构，(Multi-Layer Perceptron，MLP) 多层感知机
def mlp(x, hidden_weights, output_weights):
    hidden_outputs = tf.nn.sigmoid(tf.matmul(x, hidden_weights))
    final_outputs = tf.nn.sigmoid(tf.matmul(hidden_outputs, output_weights))
    return final_outputs
```

这里，我们定义了一个简单的多层感知机，且激活函数为 sigmoid。

- 定义训练函数

```
    # 训练
    def train():
    # 初始化权重，定义损失函数和优化器
        hidden_weights = tf.Variable(tf.random_normal([inputs, hiddens]),
name="hidden_weights")
        output_weights = tf.Variable(tf.random_normal([hiddens, outputs]),
name="output_weights")
        final_outputs = mlp(x, hidden_weights, output_weights)
        errors = tf.reduce_mean(tf.squared_difference(final_outputs, y))
    optimiser =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(errors)
    # 定义会话（session），开始训练
        init_op = tf.global_variables_initializer()
        saver = tf.train.Saver()
        with tf.Session() as sess:
            sess.run(init_op)
            total_batch = int(len(mnist.train.labels) / batch_size)
            for epoch in range(epochs):
                avg_error = 0
                for i in range(total_batch):
                    batch_x, batch_y = mnist.train.next_batch(batch_size=batch_size)
                    _, c = sess.run([optimiser, errors], feed_dict={x: batch_x, y:
batch_y})
                    avg_error += c / total_batch
                print("Epoch [%d/%d], error: %.4f" %(epoch+1, epochs, avg_error))
            print("\nTraining complete!")
            saver.save(sess, "./model")
```

- 定义预测函数

```
    # 预测
    def predict():
        saver = tf.train.import_meta_graph("./model.meta")
        with tf.Session() as sess:
            saver.restore(sess, tf.train.latest_checkpoint("./"))
            graph = tf.get_default_graph()
            hidden_weights = graph.get_tensor_by_name("hidden_weights:0")
            output_weights = graph.get_tensor_by_name("output_weights:0")
            final_outputs = mlp(x, hidden_weights, output_weights)
            correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(final_outputs,
1))
            accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```
            tf.summary.scalar('accuracy', accuracy)
            print(sess.run(accuracy, feed_dict={x: mnist.test.images, y:
mnist.test.labels}))
```

在训练时，这段 TensorFlow 代码在训练集中的误差随迭代次数（Epochs）的变化而变化，如图 2.6 所示，训练误差从第一轮迭代结束的 8.53%下降到 0.21%，到第 38 轮迭代之后，误差趋于稳定。

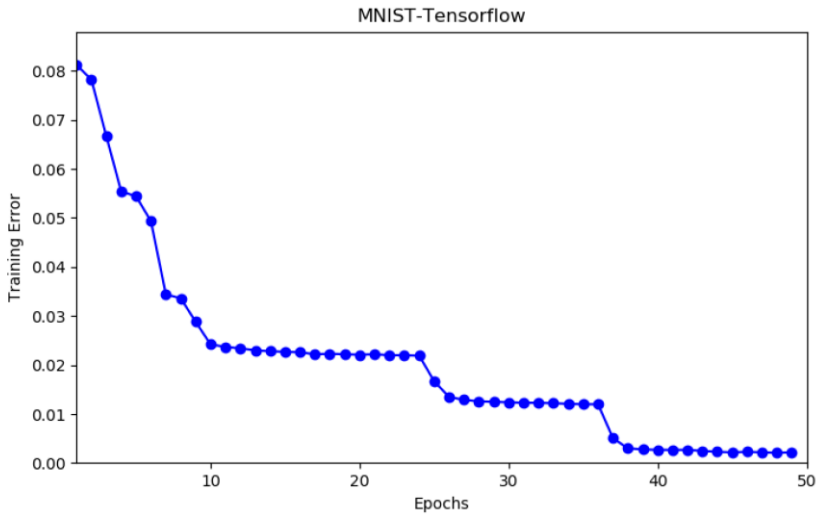


图 2.6　TensorFlow 代码的训练误差随迭代次数的变化

从上面的代码中我们能直观地感受到 TensorFlow 的符号式编程，其变量先定义成符号，譬如由 tf.placeholder 定义的 x、y 为输入符号；mlp 函数中定义的 hidden_outputs、final_outputs 为运算符号等。

在实际执行时，TensorFlow 的所有计算必须在会话（Session）里启动，所以，我们能看到在训练和测试开始之前，都包含着 with tf.Session() as sess: 这个语句。在执行时，在会话中的 x、y 会用实际的数据代入；会话会将计算图的操作分发到诸如 CPU 或 GPU 之类的设备上，同时提供执行计算图操作的方法。这些方法被执行后，将产生的张量（tensor）返回。TensorFlow 对计算图进行优化时也会对 Debug 有一些困扰。有一次，我使用 tf.Print()对一个

TensorFlow 项目进行 Debug 时一直没有输出，最后才发现是因为该数据节点不在最后输出值的执行路径上，被 TensorFlow 的计算图优化了，并没有被执行。

Keras

- 设置参数并加载数据集

```
import argparse
from keras.models import Sequential, load_model
from keras.datasets import mnist
from keras.layers import Dense
from keras import optimizers,utils

# 定义参数
inputs, hiddens, outputs = 784, 100, 10
learning_rate = 0.01
epochs = 50
batch_size = 64

# 导入数据集
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape(-1, inputs).astype('float32')/255
train_labels = utils.to_categorical(train_labels, outputs)
test_images = test_images.reshape(-1, inputs).astype('float32')/255
test_labels = utils.to_categorical(test_labels, outputs)
```

- 定义神经网络

```
# 定义多层感知机结构
def mlp():
    model = Sequential()
    model.add(Dense(hiddens, activation='sigmoid', input_shape=(inputs,)))
    model.add(Dense(outputs, activation='sigmoid'))
    return model
```

- 定义训练函数

```
# 训练
def train():
    model = mlp()
    sgd = optimizers.Adam(lr=learning_rate)
    model.compile(optimizer=sgd, loss='mean_squared_error')
    model.fit(train_images,train_labels,batch_size=batch_size,epochs=epochs)
    model.save('mlp_model.h5')
```

- 定义预测函数

```
# 预测
def predict():
    model = load_model("mlp_model.h5")
    error = model.evaluate(test_images, test_labels)
    print("accuracy:", 1 - error)
```

从语法结构上看，Keras 拥有非常简单的语法结构，简洁明了，少了如 TensorFlow“八股文”一般的会话操作，其函数更关注模型本身。在定义神经网络结构时，Keras 能够很容易地通过内置的 Sequential 函数在其上面简单地堆叠层来构建网络，每一层仅需要提供输入/输出维度和指定激活函数。在训练时，仅需要定义优化器（optimizer）、损失函数类型，并用 fit 方法来对图像和标签训练模型。最后，在训练模型后保存模型。Keras 的预测函数则更为简单，我们只需加载模型，然后使用它来评估测试图像和标签即可。

不过，Keras 底层库使用了 TensorFlow、Theano 或者 CNTK，这使得 Keras 也继承了它们的符号执行的特点，其计算也是首先定义各种变量，然后建立计算图，等到把真正需要运算的输入放入后，才能在模型中形成数据流，得到最后的输出值。Keras 的 Debug 有时也会比较麻烦，尤其是当 Debug 到底层库 TensorFlow、Theano 或者 CNTK 时。

总的来说，因为 Keras 具有简洁、模块化的特性，使得其易读、易懂、易上手，适合新手学习和使用，也适合做快速的模型迭代。

PyTorch

- 设置参数并加载数据集

```
import torch
import argparse
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
from torch.utils.data import DataLoader

# 定义参数
inputs, hiddens, outputs = 784, 200, 10
learning_rate = 0.01
```

```
    epochs = 50
    batch_size = 64

    transformation =
transforms.Compose([transforms.ToTensor(),transforms.Normalize((0.1307,),
(0.3081,))])
    train_set=datasets.MNIST('mnist/',train=True,transform=transformation,download
=True)
    train_loader=DataLoader(dataset=train_set,batch_size=batch_size,shuffle=True)
    test_set=datasets.MNIST('mnist/',train=False,transform=transformation,download
=True)
    test_loader=DataLoader(dataset=test_set,batch_size=batch_size,shuffle=False)
```

- 定义神经网络

```
    class mlp(nn.Module):
        def __init__(self):
            super(mlp, self).__init__()
            self.sigmoid = nn.Sigmoid()
            self.hidden_layer = nn.Linear(inputs, hiddens)
            self.output_layer = nn.Linear(hiddens, outputs)
        def forward(self, x):
            out = self.sigmoid(self.hidden_layer(x))
            out = self.sigmoid(self.output_layer(out))
            return out
        def name(self):
            return "mlp"
```

- 定义训练函数

```
    def train():
        model = mlp()
        loss = nn.MSELoss(reduction='sum')
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
        for epoch in range(epochs):
            avg_error = 0
            for i, (images, labels) in enumerate(train_loader):
                images = Variable(images.view(-1, inputs))
                # 将类别标签转换成 One-hot 编码
                one_hot = torch.FloatTensor(labels.size(0), 10).zero_()
                target = one_hot.scatter_(1, labels.view((labels.size(0),1)), 1)
                target = Variable(target)
                # 计算损失函数和梯度
                optimizer.zero_grad()
                out = model(images)
                error = loss(out, target)
                error.backward()
```

```
            optimizer.step()
            avg_error += error.item()

        avg_error /= train_loader.dataset.train_data.shape[0]
        print ('Epoch [%d/%d], error: %.4f' %(epoch+1, epochs, avg_error))
    torch.save(model.state_dict(), 'model.pkl')
```

- 定义预测函数

```
def predict():
    model = mlp()
    model.load_state_dict(torch.load('model.pkl'))
    correct, total = 0, 0
    for images, labels in test_loader:
        images = Variable(images.view(-1, inputs))
        out = model(images)
        _, predicted = torch.max(out.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()
    print('accuracy: %0.2f %%' % (100.0 * correct / total))
```

PyTorch 的关键特性是命令式编程，即输入什么便执行什么，非常符合 Python 的风格，也与 TensorFlow 和 Keras 的符号式编程大不相同。在使用 PyTorch 写代码时，我们可以写一行代码就执行一行，哪里出问题就停在哪里 Debug，不用像静态图那样，需要考虑到全局哪里会出问题。笔者在初次尝试 PyTorch 时，就被它的命令式编程所吸引，在尝试不熟悉的 API 时，也可以直接使用来观察结果是否符合预期。

比较特殊的是在 PyTorch 加载数据集时定义的 transformation，其中它使用 transforms.Normalize((0.1307,),(0.3081,))对数据进行正规化，0.1307 和 0.3081 分别是 MNIST 数据集的均值和方差。

MXNet-Gluon

- 设置参数并加载数据集

```
import argparse
import numpy as np
import mxnet as mx
from mxnet import nd, autograd, gluon
from mxnet.gluon import nn
from mxnet.gluon.data import vision
```

```
    # 定义参数
    inputs, hiddens, outputs = 784, 200, 10
    learning_rate = 0.01
    epochs = 50
    batch_size = 64

    ctx = mx.cpu()

    def transform(data, label):
        return data.astype(np.float32)/255, label.astype(np.float32)

    train_data = mx.gluon.data.DataLoader(vision.MNIST(train=True,
transform=transform), batch_size, shuffle=True)
    test_data = mx.gluon.data.DataLoader(vision.MNIST(train=False,
transform=transform), batch_size, shuffle=False)
```

MXNet 需要显式地指定运行环境。譬如，通过 mx.cpu()指定 MXNet 在 CPU 上运行。

- 定义神经网络

```
    def mlp():
        model = nn.Sequential()
        with model.name_scope():
            model.add(nn.Dense(hiddens, activation="sigmoid"))
            model.add(nn.Dense(outputs, activation="sigmoid"))
            dist = mx.init.Uniform(1/np.sqrt(float(inputs)))
            model.collect_params().initialize(dist, ctx=ctx)
        return model
```

在 Gluon 接口下，MXNet 定义神经网络相对简单，与 Keras 有些相似。我们只需使用内置的模型（model）添加层（layer），并使用适合的激活函数。

- 定义训练函数

```
    def train():
        model = mlp()
        loss = gluon.loss.L2Loss()
        optimizer = gluon.Trainer(model.collect_params(), "adam", {"learning_rate":
learning_rate})

        for e in range(epochs):
            cumulative_error = 0
            for i, (data, labels) in enumerate(train_data):
                data = data.as_in_context(ctx).reshape((-1, inputs))
```

```
            labels = nd.one_hot(labels, 10, 1, 0).as_in_context(ctx)
            with autograd.record():
                output = model(data)
                error = loss(output, labels)
            error.backward()
            optimizer.step(data.shape[0])
            cumulative_error += nd.sum(error).asscalar()
        print("Epoch [%d/%d]: error: %.4f" % (e+1, epochs,
cumulative_error/len(train_data)))
    model.save_parameters("mxnet.model")
```

- 定义预测函数

```
def predict():
    model = mlp()
    model.load_params("mxnet.model", ctx)
    acc = mx.metric.Accuracy()
    for i, (data, label) in enumerate(test_data):
        data = data.as_in_context(ctx).reshape((-1, inputs))
        label = label.as_in_context(ctx)
        output = model(data)
        predictions = nd.argmax(output, axis=1)
        acc.update(preds=predictions, labels=label)
    print("accuracy: %.2f %%" % (acc.get()[1] * 100))
```

对于 MXNet，我们单独列出了其 Gluon 接口。笔者体验过使用 Gluon 接口前后的 MXNet，采用了 Gluon 接口后的 MXNet，其编程效率确实高了许多。Gluon 接口最大的改变是将 API 接口变得简洁，其简洁程度介于 TensorFlow 与 Keras 之间，与 PyTorch 类似。MXNet-Gluon 支持混合编程，即同时支持命令式编程和符号式编程。在编写新模型时，可以使用命令式编程，方便调试；在实际模型上线时，可以比较容易地转变为符号式编程，能进行更多优化。

在实际中，使用 MXNet-Gluon 编写深度学习模型，比较容易上手，代码简洁。有一些遗憾的是，MXNet 可以参考的样例代码略少，有些 API 并不能使用。其中的一个原因是，MXNet 在文档详细程度和社区规模上相比 TensorFlow、PyTorch 等还是偏弱，希望以后 MXNet 能在文档和社区上做得更好。

CNTK

- 设置参数并加载数据集

```
    import argparse
    import numpy as np
    import cntk as C
    from cntk.ops.functions import load_model
    from cntk.io import MinibatchSource, CTFDeserializer, StreamDef, StreamDefs

    # 定义参数
    inputs, hiddens, outputs = 784, 200, 10
    num_hidden_layers = 1
    learning_rate = 0.01
    epochs = 50
    batch_size = 64

    np.random.seed(0)
    # 选择合适的设备（此处使用 CPU）
    C.device.try_set_default_device(C.device.cpu())

    # 定义 CTF 格式数据读取函数
    def create_reader(path, is_training, inputs, num_labels):
        return MinibatchSource(CTFDeserializer(path, StreamDefs(
            labels= StreamDef(field='labels', shape=num_labels, is_sparse=False),
            features= StreamDef(field='features', shape=inputs, is_sparse=False)
        )),     randomize= is_training, max_sweeps= C.io.INFINITELY_REPEAT if
is_training else 1)
    # 导入训练集和测试集数据
    train_file = "./data/MNIST/Train-28x28_cntk_text.txt"
    reader_train = create_reader(train_file, True, inputs, outputs)
    test_file = "./data/MNIST/Test-28x28_cntk_text.txt"
    reader_test = create_reader(test_file, False, inputs, outputs)

    input = C.input_variable(inputs)
    label = C.input_variable(outputs)
```

- 定义神经网络

```
    def mlp(features):
        with C.layers.default_options(init = C.layers.glorot_uniform(), activation =
C.ops.sigmoid):
                h = features
                for _ in range(num_hidden_layers):
                    h = C.layers.Dense(hiddens)(h)
                r = C.layers.Dense(outputs, activation = None)(h)
                return r
```

- 定义训练函数

```
    def train():
        # 将数据范围缩放为 0~1
        model = mlp(input/255.0)
        loss = C.cross_entropy_with_softmax(model, label)
        errors = C.classification_error(model, label)
        lr = C.learning_rate_schedule(learning_rate, C.UnitType.minibatch)
        learner = C.adam(model.parameters, lr, momentum=0.9)
        trainer = C.Trainer(model, (loss, errors), [learner])
        # 映射数据和标签
        input_map = {
            label: reader_train.streams.labels,
            input: reader_train.streams.features
    }
        train_num_samples = 60000
        # Run the trainer on and perform model training
        total_batch = int(train_num_samples / batch_size)
        for epoch in range(epochs):
            avg_error = 0
            for i in range(total_batch):
                data=reader_train.next_minibatch(batch_size, input_map=input_map)
                trainer.train_minibatch(data)
                avg_error += trainer.previous_minibatch_evaluation_average /
total_batch
            print("Epoch [%d/%d], error: %.4f" %(epoch+1, epochs, avg_error))
        model.save('outputs/cntk.model')
```

- 定义预测函数

```
    def predict():
        model = load_model('outputs/cntk.model')
        test_input_map = {
            label: reader_test.streams.labels,
            input: reader_test.streams.features,
        }
        eval_num_samples = 10000
        data = reader_test.next_minibatch(eval_num_samples, input_map =
test_input_map)
        pred = np.argmax(model.eval(data[input]), axis=1)
        gtlabel = np.argmax(data[label].asarray(), axis=2).flatten()
        accuracy = (np.equal(pred, gtlabel).sum())/float(eval_num_samples)
        print("accuracy: %.2f %%" % (accuracy * 100))
```

在我们尝试的几个深度学习框架中，CNTK 的实现难度最大。其代码比较冗

余，数据处理也不大方便。譬如上面介绍的 CNTK 代码的数据读入部分比较复杂，其原因是 CNTK 并没有提供直接读取 MNIST 数据集的 API，它需要先下载 MNIST 数据集，并利用其 Git 代码库中的 CNTK\Examples\Image\DataSets\MNIST\install_mnist.py 文件进行数据格式转换，转换成 CNTK 的文本格式。运行 install_mnist.py 之后，会生成两个文件 Train-28x28_cntk_ text.txt 和 Test-28x28_cntk_text.txt，即本节中 CNTK 脚本所用的文件。

结合 2.1 节中的表 2.1，我们能够看出一些端倪，CNTK 的贡献者数量仅有 191 人，远少于其他开源框架的贡献者，譬如 TensorFlow 有 1817 位贡献者。从 2018 年 11 月开始，微软对 CNTK 的政策从积极支持转为继续维护。CNTK 未来的发展，恐怕也要步 Theano 的后尘了。

最后，来看一看以上介绍的几个开源框架的代码在预测准确率和训练时间上的差异。其中实验环境是 Windows 10，Intel Core E5-1650 v4，时钟频率为 3.6GHZ，内存为 32GB RAM，三级缓存，分别是 32KB、256KB 和 15MB。实验结果如表 2.2 所示。

表 2.2　开源框架的代码的预测准确率和训练时间

开源框架名称	TensorFlow	Keras	PyTorch	MXNet	CNTK
预测准确率	97.31%	99.45%	98.00%	97.77%	97.56%
训练时间	55s	1m15s	4m38s	3m11s	1m12s

总的来看，除 Keras 的测试准确率达到了 99.45%外，其他几个框架的测试准确率都很接近，准确率在 97.5%~98%之间。Keras 有更高的准确率是因为其在 MNIST 数据集上对参数的初始化更合理。

在训练时间上，我们看到 TensorFlow、Keras 和 CNTK 比较接近，要快于 PyTorch 和 MXNet。这个差异主要来自命令式编程与符号式编程的不同，采用符号式编程的框架运行速度能比采用命令式编程的框架快数倍。但另一方面，命令式编程方便调试，能减少编写代码的时间。在小规模的数据集上，编写代码的时间远远大于训练时间。读者可以结合实际的项目来选用合适的框架。

2.3 本章小结

深度学习框架一直在迅猛发展，在写作本书的过程中，PyTorch 的版本从 0.4.1 版更新到 1.0 版，CNTK 的版本从 2.5.1 版更新到 2.6 版，MXNet 的版本从 1.3.0 版更新到 1.3.1 版，TensorFlow 的版本从 1.10 版更新到 1.12 版，Keras 的版本从 2.2.2 版更新到 2.2.4 版。这样的发展速度使得严格地比较各个框架比较困难。随着版本的更新，缺点被克服，优点在发扬，不同的框架都在发展得越来越好。

本章主要试图比较不同深度学习框架在代码上的不同，并且让读者弄清楚编写深度学习模型的难易程度。从实际上手情况来看，深度学习模型的编写难易程度主要和框架本身及文档的详细程度有关：在框架设计上，Keras 最为简洁，原因是 Keras 更偏上层，需要编写的代码量更少，其他框架需要编写的代码量的差别不大；文档详细程度在一定程度上也和社区规模相关，譬如 TensorFlow，其拥有的社区用户数量最多，并且文档齐全，开源项目也很多。

这些框架的目标基本一致，即在每个框架中，用一个简单的方法来加载数据集、定义模型、训练模型，然后使用它来预测结果。虽然具体实现的方式可能有所不同，从框架到框架，潜在的设计哲学可能也会有所不同，但目标基本保持一致。随着深度学习的发展，其中的一些最佳实践被总结和保留了下来。所以从代码上看，同一个深度学习任务在不同机器学习框架下，流程结构大体相同。这对用户来说无疑是福音，从一个机器学习框架迁移到另一个机器学习框架不需要太多的学习成本。

对于深度学习框架的选择，因人而异。这些机器学习框架的差异在于设计理念、支持的平台、支持的语言等。用户可以根据自己的目的选择最适合的框架。对初学者来说，可能使用代码简洁的 Keras 或者开源项目丰富的 TensorFlow 会是一个比较好的开始；想要调试方便，PyTorch 也是一个不错的选择。

相信今后的深度学习框架在朝着以下相同的目标努力。

（1）兼容与开放。长期以来，不同深度学习框架互不兼容，一个深度学习框架产生的模型并不能在另一个深度学习框架上使用。ONNX 的出现，让模型在一个深度学习框架上训练，在另一个框架上预测成为可能。

（2）越来越快的速度。在速度方面，越来越快是趋势，分布式、C++后端、GPU、CUDA、AVX、MPI 等技术的出现和应用，使得训练大规模的深度学习模型越来越容易。这有助于更多的深度学习模型产品化，面向用户。

（3）简洁明了的语法。提供简洁的语法方便编写深度学习算法，能够大大提高程序员的工作效率。例如分析历史趋势，在 Theano 出现之前，编写深度学习算法需要写许多行代码。Theano 对数值计算的分装，解决了重复“造轮子”的事情，所以其流行起来。此后，TensorFlow、PyTorch、MXNet、CNTK 继承了 Theano 的优点，更为简洁明了。而目前基于框架的框架 Keras 的势头在蒸蒸日上，也是因为其方便，代码简洁易懂。在未来，深度学习框架一定会往简洁、易用的方向发展，让科研人员和工程人员能聚焦于深度学习算法本身。

参考文献

[1] ABADI M, BARHAM P, CHEN J M, et al, Tensorflow: A system for large-scale machine learning. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI16) 2016 [C], 265-283.

[2] JIA Y Q, SHELHAMER E, DONAHUE J, et al, Caffe: Convolutional architecture for fast feature embedding. Proceedings of the 22nd ACM international conference on Multimedia 2014 [C], ACM, 675-678.

Deep Learning Models and Applications

第 3 章

多层感知机在自然语言处理方面的应用

多层感知机是一类前馈神经网络，是最先被发明也是最简单的一类神经网络。所谓前馈神经网络，是指神经网络节点（node）之间的连接（connection）不存在环。即在前馈神经网络中，前一层节点的输出会作为后一层节点的输入，这样一层层计算直到最后输出层输出结果。在此过程中，不存在后一层节点的输出反馈给前一层节点作为输入的情况。多层感知机、卷积神经网络等均属于前馈神经网络。多层感知机是至少含有一层隐藏层的前馈神经网络，且其层与层之间是全连接的。与前馈网络相对的是递归神经网络。在递归神经网络中，网络中的某一层节点的输出会反馈给之前的网络层作为输入进行下一步计算，即网络节点之间的连接会形成环，因此，递归神经网络不属于前馈神经网络。

【知识点讲解】虽然从结构上讲，递归神经网络中的网络节点存在反馈环，但是如果将递归神经网络中反馈的部分进行展开，则可以将其表示为共享相同参数的多层前馈神经网络。这种展开形式往往便于我们直观理解递归神经网络的具体计算流程。展开的具体方法会在第 5 章进行介绍。

3.1 词和文本模型的发展历程

在信息检索、搜索广告等多个领域中，如何对词及文本（例如用户输入的查询、互联网网页中的内容、广告主投放的广告标题及描述）进行建模并进而计算其相似度，是一个非常根本而且关键的问题。在具体介绍词的分布式表示模型之前，先回顾一下在信息检索这个领域中，词以及文本模型的发展历程。这样我们就可以比较清楚地理解，词的分布式表示模型有哪些关键优势，解决了哪些关键问题。

在 20 世纪 60 年代，bag-of-words 模型用于对文档进行建模。在这个模型中，每个词都用高维空间中的某一个特定维表示，这样一个词就可以用一个只有某一维为 1 而其他维为 0 的向量（one-hot-vector）来表示。进而，一个文档可以用它所包含词的词向量加和来表示，得到文档向量。直观地讲，这个文档向

量是一些维为正值而另外一些维为 0 的高维向量。它表示，取值不为 0 的那些维度对应的词出现在了当前文档中，而取值为 0 的那些维度对应的词没有出现在文档中。通常，我们可以计算两个文本向量的余弦相似度的值来度量它们的相似程度。

基于上述模型，有各种各样的改进和变形使得文档相似度的计算越来越准确。例如在 20 世纪 70 年代提出的 TF-IDF 算法，改进了词权重的计算方法，大幅度提高了文档相似度计算的准确率，并得到广泛应用。具体来讲，在计算文档向量时，不再用简单的 0/1 来表示是否含有这个词，而是将 1 替换为一个权重来表示词的重要程度。当然，对于文档中并未出现的词，仍然以 0 作为对应维的值。在一个文档中，对于任意一个词，以这个词在当前文档中出现的次数(term frequency)为分子，以这个词在所有文档中出现的次数的 log 值为分母(document frequency)，把分子和分母相除得到的商作为这个词对应的权重。其表达的含义是：把所有文档放在一起考虑，那些文档中较少出现的词往往表达了更加具体、更加有针对性的含义，因此，在文档相似性度量中应给予较高权重；而在大部分文档中都出现的词，表达的含义往往比较宽泛，因此，在文档相似性度量中应给予较低权重。随后，基于 TF-IDF 算法类似的思路，又出现了诸多对于文档相似度函数的各种改进和变形，例如 BM25。

虽然有诸多变形和改进，但它们几乎仍是基于 one-hot-vector 来对词进行表示的，只是在如何计算词的重要程度时有所不同。one-hot-vector 即是将不同的词用高维向量中的不同维度表示，其有一个根本的缺点，那就是它无法表达词的语义相关度。例如，“称赞”和“夸奖”是一对近义词，它们往往出现在相似的语境中，表达相似的意思；反之，“称赞”和“诋毁”是一对反义词，表达的意思截然相反；此外，“称赞”和“营业”是两个语义无关的词，表达的语义没有关联。但是在使用 one-hot-vector 这种词表示模型时，这些词各自对应着高维空间的某一个特定的维，它们彼此之间的相似度都一样，均为 0。因此，

one-hot-vector 这种词表示模型不能够体现词的语义信息。也是因为这个原因，基于此的 bag-of-words 文档模型在本质上也仅是依靠计算不同文档中重合的词的比重，并以此来衡量文档的相关度的，不能够很好地体现我们所真正关心的语义相关度。TF-IDF 算法及其变种虽然根据词频对词的重要程度进行了建模，但是仍不能反映其语义相关度。而基于多层感知机实现的词的分布式表示模型通过对词的语义进行编码，进而解决了 one-hot-vector 这种词表示模型对语义表示不足的缺陷。下面就介绍这个近年来在自然语言处理领域得到广泛应用的技术。

3.2 Word2Vec 模型：基于上下文的分布式表达

在分布式表示模型下，一个词被映射为一个高维向量。与 one-hot-vector 不同的是，这个向量并非只有一维为非 0，而是各个维均有其对应的值，表示其在对应的语义子空间的语义分量。对于两个词，我们可用其对应的分布式嵌入向量之间的余弦相似度（cosine similarity），进而体现其语义相关度。此外，对于两个近义词，其对应的分布式嵌入向量往往比较接近。更奇妙的是，这种分布式表示向量往往还符合类比的性质。例如我们可以做如下类比：国王是男人，王后是女人。而后可以发现，$\boldsymbol{e}_{\text{queen}} - \boldsymbol{e}_{\text{woman}} \approx \boldsymbol{e}_{\text{king}} - \boldsymbol{e}_{\text{man}}$。其中$\boldsymbol{e}$为词的分布式嵌入向量。这些都充分说明了分布式表示模型可以对词的语义进行建模和表示。

那么，什么是语义呢？而我们又如何对词的语义进行建模呢？在基于统计的自然语言处理中，一个关键且核心的观点就是，词的语义可以由其所处的上下文（context）所表达。基于这个观点，我们可以把词的语义建模转换为对词的上下文的建模。具体来说，在给定的一个语料库中，对于任意一个给定的词，我们都可以统计这个词在语料库中出现时，其对应的上下文（例如其前后各 3 个词）。这样，我们就可以将其上下文中的所有词的分布，作为这个词的语义的一种表达。

【知识点讲解】严格地讲，一个词的上下文不能完全表达出这个词的语义信息，否则这个词就可以被省略了。但是，用词的上下文来表示词的语义，是一种

比较合理的近似。具体来说，如果两个词对应的上下文信息非常类似，那么可以认为这两个词具有相似的语义。例如，“称赞”和“夸奖”经常出现在类似的上下文语境中，因此，它们对应的上下文比较类似，也可以说它们具有相似的语义。用词的上下文来表示这个词的语义，既直观，也将语义建模这个抽象的概念变得可以操作。

Word2Vec 模型也是基于这个核心观点的，它的基本思想为：首先用一个神经网络（映射网络），将词映射为一个向量（即分布式嵌入向量）；然后将这个向量通过另外一个神经网络（预测网络）生成一个词分布，使得这个生成的词分布与其对应的上下文分布尽可能一致。由于分布式嵌入向量通过预测网络可以近似表达词的上下文分布，因此，我们认为分布式嵌入向量包含了词的上下文信息，进而也就可以作为词语义的一种合理表达。

下面介绍一下 Word2Vec 的具体算法。目前，Word2Vec 有两种最常见的算法，分别是 CBOW（Continuous Bag-of-Words）算法和 Skip-Gram 算法。这两个算法有许多相似之处，它们互为镜像，理解了其中一个算法，另外一个算法也就非常容易理解了。下面主要介绍 Skip-Gram 算法。

3.2.1 Skip-Gram 算法的训练流程

Skip-Gram 算法的基本训练流程如下：在给定的数据集上，对于每一个词，得到其上下文。然后将当前词作为输入，其对应的上下文作为输出，训练神经网络。训练的目标是网络的输出（即预测的上下文）和其实际的上下文越接近越好。

以如下数据集为例子，对训练流程进行具体介绍。假设数据集中有这样的一句话：“I go to the playground to play football with my classmates”。对于数据集中的每一个词，都将生成一条训练样本，其输入是当前词，输出是其对应的上下文。我们可以通过任何合理的方式定义词的上下文。通常的做法是把一个词前后的几个词作为它的上下文，而具体取几个词是可以通过参数设置的，这个参

数叫作上下文窗口大小。如果上下文窗口大小为 1，则意味着我们将这个词的前一个词以及后一个词作为它的上下文；如果上下文窗口大小为 2，则把这个词的前后各两个词作为它的上下文；依此类推。在实际应用时，上下文窗口大小的具体取值根据应用场景的不同而不同，但一般来说不超过 5。对于上面的例子，如果将上下文窗口大小设为 1，则将得到如下的样本集合，{(I,[Null,go]),(go,[I,to]),(to,[go,the]),(the, [to,playground]),…,(classmates,[my,null])}。其中，每个小括号内是一条样本的输入和输出，用逗号分隔。中括号内是输出，对应着当前词的上下文。请记住，在训练 Skip-Gram 模型时，我们将当前词作为输入，而将它的上下文作为输出。例如，对于样本(go,[I,to])，我们的训练目标是用词“go”预测得到词“I”以及“to”；对于样本(classmates,[my,null])，我们的训练目标是用词“classmates”预测得到词“my”。在这里，我们忽略了句子首、尾处超出“窗口”的部分。

为了处理方便，我们进一步将每条样本中的输出部分（即上下文）进行分拆，从而将上述样本集合转换为如下集合{(I,go),(go,I)(go,to),(to,go)(to,the),(the,to),(the,playground),…,(classmates,my)}。这样，基于给定的一个语料数据集，我们就可以根据这种方式构建训练样本集合。

需要指出的是，在上述根据语料数据集构建训练样本集合的过程中，我们无须预先对语料数据集中的数据进行任何标注，只利用其上下文关系来构建训练样本集合。利用语料自身的上下文信息而非特别的人工标注就可以构建训练样本集合，是一个极大的优势。这是因为我们往往可以非常容易地收集到这种无须特别标注的语料数据集，因此能够以较低的成本和代价构建规模很大的训练样本集合，而大规模的训练样本集合往往对于提高模型性能有非常大的帮助。

另外需要说明的是，我们将各个词与其各自上下文中所含的词，构成的输入/输出对，仅仅构成了正样本集合。例如(football, play)是一条正样本，表示单词“play”出现在“football”的上下文中。但是在训练模型的时候，为了避免模型

被训练成为永远输出正类的平凡分类器，我们也需要负样本。从理论上讲，对于一个给定的词，例如“football”，所有没有出现在其上下文中的词和“football”组成的词对均应被视为负样本来进行训练。在我们的例子中，除“play”和“with”外的词，和“football”组成的词对均是负样本，例如(I ,football)，(go，football)等。假设语料库中共有V个不同的词，如果我们在构建负样本时考虑所有的词（当然，需要去除上下文中所含的少数词），那么训练的计算量将会近似增长V倍。在实际的应用中，语料数据集往往很大，V的取值可以达 1000 万级别或更多。因此，这样构建负样本的做法将会使得训练的计算量变得过大，进而使得训练变得非常困难。

为了解决上述问题，有各种各样的方法进行近似计算以降低复杂度。Tomas[1]等人应用了层级结构（具体可采用霍夫曼编码）的设计将语料库中的所有词进行编码，将计算量从V倍降到$\mathrm{Log}_2(V)$倍。但现在最为普遍的做法是使用负样本采样方法(negative sampling)，即对每一个正样本，利用随机采样构建n个负样本。其中n是一个可配置的参数，可根据具体情况设置（例如n=4）。仍以前述单词为例，对于“football”，虽然除“play”和“with”外的词都可以和“football”组成负样本，但是我们只是在这些词中随机取n（例如n=4）个词出来和“football”组成负样本词对。负样本采样方法可以有效降低训练的运算复杂度，且有理论分析证明了其合理性和有效性。当数据集的规模足够大时，采取 Skip-Gram 算法结合负样本采样，是效果最好的做法，也是在实际中应用最广的做法。

【知识点讲解】CBOW（Continuous Bag-of-Words）算法和 Skip-Gram 算法互为镜像。在训练时，对于每一个词，它以该词的上下文作为输入，而将该词作为输出，即用词的上下文来预测该词。而前述的 Skip-Gram 算法则恰恰相反，

1 MIKOLOV T, SUTSKEVER I, CHEN K, et al. Distributed representations of words and phrases and their compositionality. Proceedings of the 26th International Conference on Neural Information Processing Systems 2013[C], 3111-3119 .

是用当前词来预测其上下文的。

3.2.2 Skip-Gram 算法的网络结构

当网络结构的上下文窗口大小为 2 时，Skip-Gram 算法的网络结构如图 3.1 所示。其中x_t表示第t个词，x_{t-1}和x_{t-2}分别表示第t个词的前面第一个词和前面第二个词；x_{t+1}和x_{t+2}分别表示第 t 个词的后面第一个词和后面第二个词。x_t经过一个编码映射（encoder），得到一个d维的列向量e_t，被称为x_t的分布式嵌入向量（embedding vector）。而这个分布式嵌入向量，进一步经过一个解码映射（decoder），输出对x_t上下文的预测。这个网络的训练目标就是使得网络预测输出的上下文分布和训练样本中真实的上下文分布尽可能接近。

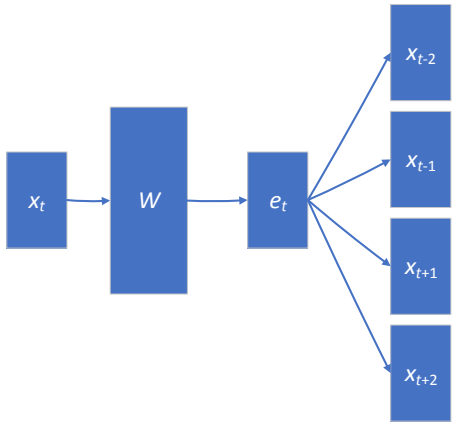


图 3.1 Skip-Gram 算法的网络结构图

1. 输入层

假设整个训练数据集共有V个不同的词，每个词对应一个取值范围在$[0, V-1]$的序号。图 3.1 中的可以用一个V维的 one-hot-vector 向量表示，那它对应序号维度的取值为 1，而其他维度的取值均为 0。

2. 编码映射层

在编码映射层，对于输入词序列的每个词x_t，编码映射层将其

one-hot-vector 通过如下的线性变换映射为一个低维空间的向量$\boldsymbol{e}_t$，被称为分布式嵌入向量。

$$\boldsymbol{e}_t = \boldsymbol{W}_1\boldsymbol{x}_t + \boldsymbol{b}_1, \text{ s.t. } \boldsymbol{W}_1 \in \mathbf{R}^{d\times V}, \boldsymbol{b}_1 \in \mathbf{R}^{d\times 1} \quad (1)$$

其中，$\boldsymbol{W}_1$和$\boldsymbol{b}_1$分别是编码映射层的变换矩阵及偏置向量。请注意，$\boldsymbol{x}_t$是一个one-hot-vector，如果它的第i维为 1，则$\boldsymbol{W}_1\boldsymbol{x}_t$的结果是矩阵$\boldsymbol{W}_1$的第$i$列。

3. 解码映射层

前一编码映射层产生的输出，将作为后续解码映射层的输入。其具体的运算公式如下：

$$\boldsymbol{z}_t = \boldsymbol{W}_2\boldsymbol{e}_t + \boldsymbol{b}_2;\ \boldsymbol{h}_t = \sigma(\boldsymbol{z}_t) \text{ s.t. } \boldsymbol{W}_2 \in \mathbf{R}^{V\times d}, \boldsymbol{b}_2 \in \mathbf{R}^{V\times 1} \quad (2)$$

其中，$\boldsymbol{W}_2$和$\boldsymbol{b}_2$是解码映射层的变换矩阵及偏置向量，$\boldsymbol{z}_t$对应着中间输出结果，$\sigma()$是 softmax 函数。经过 softmax 函数映射，我们得到的$\boldsymbol{h}_t$是一个维度为V，各维取值为(0,1)的向量，且$\boldsymbol{h}_t$的各个维的取值之和为 1。$\boldsymbol{h}_t$作为网络的输出，表示了预测的$\boldsymbol{x}_t$的上下文词的概率分布。我们的目标是，预测的上下文词的概率分布，与训练样本集上的上下文词的分布尽可能一致。

【知识点讲解】softmax 函数是将一个高维向量$\boldsymbol{x}$（$\boldsymbol{x} \in \mathbf{R}^{V\times 1}$）映射为同维度的向量$\boldsymbol{y}$（$\boldsymbol{y} \in \mathbf{R}^{V\times 1}$），且$\boldsymbol{y}$的各维取值为(0,1)，各维取值和为 1。softmax 的函数形式为：$y_j = \frac{\exp(x_j)}{\sum_{k=1}^{V}\exp(x_k)}$。其中$x_j$和$y_j$分别表示输入向量$\boldsymbol{x}$和输出结果向量$\boldsymbol{y}$的第$j$维。softmax 函数是一种在多类分类别问题中普遍使用的函数映射形式，其作用是将一个高维向量转换为一个概率分布。

3.2.3 代价函数

如前所述，将当前词$\boldsymbol{x}_t$作为网络输入，结果如图 3.1 所示，最终得到网络的输出$\boldsymbol{h}_t$，它表达了当前网络对于$\boldsymbol{x}_t$上下文的词分布的预测。我们的目标是，对于训练集上的所有词的各自上下文分布的预测，和训练样本集上真实的上下文的词

分布，尽可能一致。那么，如何具体用严格的数学公式来刻画“一致”的程度呢？这就需要定义代价函数。代价函数是用严格的数学公式来刻画预测分布与真实分布的差别。其差别越小，表明一致程度越高，说明训练得到的模型越准确。

【知识点讲解】在构造代价函数时，需要考虑以下几个因素：

（1）代价函数必须能够比较好地刻画和体现我们所期望的实际目标。只有这样，通过优化代价函数而得到的模型，才能和实际目标相契合，取得较好的实践效果。举例来说，如果我们希望预测一只股票的价格，实际目标是预测的偏差越小越好，则对应地，我们就可以均方误差作为代价函数。由于这个代价函数很好地刻画了预测偏差这个实际目标，因此，通过最优化这个代价函数而得到的模型，与实际目标是相契合的。

（2）在构造代价函数时，还需要考虑它是否易于优化。仍以股票价格预测为例，其目标仍是预测的偏差越小越好。此时，所谓的偏差，既可以用平方误差来表示，也可以用绝对值误差来表示，这就对应了两种不同的代价函数。这两种代价函数都可以较为合理地刻画实际目标。但是，由于均方误差是可微分函数，便于应用各种基于导数的优化算法，因此，使用均方误差的代价函数比使用绝对值误差的代价函数应用更加广泛。

根据我们的目标，即刻画预测的上下文词分布与真实上下文词分布的一致性，我们可以采取交叉熵（cross entropy）作为代价函数，这是一种刻画两个分布之间的差异的常用代价函数。它的具体形式如下，其中，$p(x)$和$q(x)$是两个分布函数。

$$H(p,q)=\sum_i p(x_i)\log\frac{1}{q(x_i)} \tag{3}$$

我们以前面即语料数据集“I go to the playground to play football with my classmates”为例，讲解交叉熵这个代价函数在 Skip-Gram 算法中具体是如何计算的。如前所述，我们将语料数据转换为多个词对，构成训练数据集：{(I, go),

(go,I) (go,to),(to,go)(to,the),(the,to),(the,playground),…, (classmates,my)}。我们对各个词按字母顺序进行索引编号（见表 3.1）。

表 3.1　将词索引编号

索引编号	0	1	2	3	4	5	6	7	8	9
词	classmates	football	go	I	my	play	playground	the	to	with

对于任意一条训练样本，例如(play,football)，它从输入到输出的计算流程如图 3.2 所示，其中$\sigma()$是公式（2）中的 softmax 函数。对于当前这一条训练样本，其代价函数按照公式（3）计算如下。

$$J_{\mathrm{k}}(\theta)=-\sum_{i=0}^{9}p(y^i)\log(h^i)=-\log(h^1)=-\log\frac{\exp(z^1)}{\sum_{i=0}^{9}\exp(z^i)} \tag{4}$$

其中，θ代表了模型的参数集合，即$\{\boldsymbol{W}_1,\boldsymbol{b}_1,\boldsymbol{W}_2,\boldsymbol{b}_2\}$。$J_k(\theta)$中的下标$k$表示这是第$k$个训练样本。$y^i$、$h^i$以及$z^i$的上标$i$分别表示目标向量$\boldsymbol{y}$、预测向量$\boldsymbol{h}$，以及中间输出向量$\boldsymbol{z}$的第$i$维，$i\in\{0,1,\cdots,9\}$。

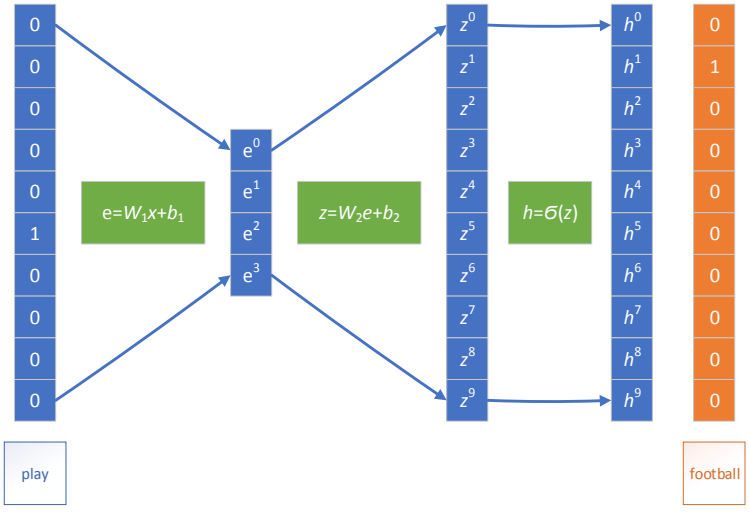


图 3.2　训练样本示例

请注意，在公式（4）中，我们需要计算这一项 $\sum_{i=0}^{9}\exp(z^i)$，其中包括了训

练样本集中所有词的中间输出结果。在这个简单的说明性例子中，总共只有 10 个不同的词，所以求和项中只有 10 项。但是在实际的应用中，词表中词的个数往往可以达到数十万的量级或者更高。这时，公式（4）分母中的求和项的运算复杂度将会变得非常巨大。为了解决这个问题，Word2Vec 模型的作者采用了负样本采样技术进行简化的近似计算。上述具体示例的代价函数，采用负样本采样技术进行简化后，可转换为公式（5），具体的算法原理和分析，读者可参考相关资料 。[1]其中，$i_n \in \{0,1,\cdots,9\}$，且$i_n \neq 1$。它是从词表中随机抽取的不属于“play”上下文的词。公式（5）的直观含义是：对于样本的输入词“play”，它的上下文词“football”是正样本；而词表中未出现在它上下文中的词，是负样本，例如“go” “classmates”等。我们希望正样本对应的中间结果（在这一条训练样本中，“football”对应的中间结果是z^1）尽可能地大，而负样本对应的中间结果（例如“go”和“classmates”对应的中间结果是z^2和z^0）尽可能地小。由于在实际应用中，负样本的数量特别巨大，我们仅随机抽取其中的n个来参与计算，因此得到公式（5）来定量刻画这个目标。事实上，这个代价函数可以非常好地近似严格的交叉熵，且能够大大降低训练的复杂度，提高模型训练的效率。

$$J_k(\theta) = \log(1 + \exp(-z^1)) + \sum_n \log(1 + \exp(z^{i_n})) \qquad (5)$$

公式（5）是针对具体的一条训练样本计算得到的代价函数。对于训练样本集上的每一条训练样本，我们都可以类似地计算其各自的代价函数。模型在训练样本集上的整体代价函数就是其在各个训练样本上的代价函数之和。而训练的目标是找到最优参数θ_{opt}，使得模型在训练样本集上的整体代价函数最小，公式如下：

1 MIKOLOV T, SUTSKEVER I, CHEN K, et al. Distributed representations of words and phrases and their compositionality. Proceedings of the 26th International Conference on Neural Information Processing Systems 2013[C], 3111-3119.

$$\theta_{\text{opt}} = \text{argmin}\big(J(\theta)\big) = \text{argmin}(\sum_k J_k(\theta)) \quad (6)$$

3.3 应用 TensorFlow 实现 Word2Vec 模型

本节介绍如何用 TensorFlow 实现前面介绍的 Word2Vec 模型(Skip-Gram 算法 + Negative Sampling)。

在具体介绍代码之前，我们需要对 TensorFlow 的一些核心概念和基本工作原理有所了解。TensorFlow 最初是源于 Google 内部使用的一个深度学习框架。在 2015 年年底，Google 开源了 TensorFlow，使得所有的深度学习工作者都可以利用这个机器学习框架进行模型开发。

事实上，TensorFlow 的功能不局限于深度学习，它的本质功能是定义和实现一个计算图。因此，任何一个可以抽象为计算图模型的问题，都可以用 TensorFlow 来解决。当然，目前 TensorFlow 最主要的应用仍然是在深度学习领域。

在使用 TensorFlow 实现机器学习模型的训练时，往往遵循如下基本步骤。

第一步，定义计算图。即先定义一个从原始样本输入到最终输出的计算流程。简单地说，就是实现一个算法，定义如何从样本的原始输入变量，通过各种各样的神经网络计算单元，得到最终的模型输出结果。而这个计算流程本身就是神经网络的前向预测模型。在计算流程中所涉及的各种各样的参数，就是待优化的模型参数。TensorFlow 会把这个模型（也即计算流程）自动转换为对应的计算图并存储起来。TensorFlow 中有大量的预定义的数学运算符（例如 max、add、mul），各种各样常见的神经元，以及搭建常见神经网络所需要的操作及运算。通过这些预定义的运算符及函数，我们可以很方便地定义从输入到输出的计算流程。特征学习和提取是深度神经网络不同于传统机器学习算法的关键所在，各种各样深度学习算法的区别往往也都体现在如何将原始输入样本变换为特征向量这一关键步骤上。

第二步，基于模型的输出及预测目标定义代价函数，并指定使用的优化算法优化此代价函数。对于监督学习问题和非监督学习问题，其代价函数的定义有较大的不同，因此这里分开讲解。对监督学习来说，这一步往往就是将特征向量映射为某一个类别，并得到其属于这个类别的置信度。有了样本的预测类别及其对应的置信度，并结合样本对应的标注信息，我们就可以进一步定义代价函数。这个代价函数往往反映了预测结果和真实标注结果的差异。最大熵是常见的优化代价函数准则。而模型训练的目标是，通过调整模型中所有可训练的参数，使得学习完毕后所得到的最终模型在训练集上对应的代价函数值最优（即预测结果和真实标注结果差异最小）。

对非监督学习来说，由于问题的特点和目标往往有很大的不同，因此，这一步的具体做法也有较大的差异。以 Auto-Encoder 为例，在这一步，它是将特征向量映射回原始样本的输入空间，即进行样本重构，然后将重构样本与原始输入样本进行比较，计算其差异，并将重构误差作为代价函数。而对 K-means 聚类问题来说，在这一步，它是将特征向量匹配到最近的一个聚类，并计算特征向量到此聚类中心的距离。最终的代价函数是各样本到其对应聚类中心的距离的平方之和。

尽管问题千差万别，但无论是什么问题，我们都需要使代价函数能客观、合理地反映问题的目标。

有了代价函数作为优化目标，我们还需要定义训练模型所使用的优化算法，来具体学习和更新模型参数。在 TensorFlow 中，有大量的已经定义好的优化算法可供直接使用。在大部分情况下，你并不需要自己去实现一个优化算法。

第三步，运行优化算法学习模型参数。具体来说，前两步（定义计算流程图及代价函数）都是静态的，而第三步则是开始计算，通过一轮轮的优化及迭代完成模型的训练。首先是读取输入数据，然后运行第一步中定义的计算流程图，并计算第二步中定义的代价函数。之后，运行相应的优化算法以更新模型参数，直

到模型收敛完成模型参数的学习。

下面以 Word2Vec 模型为例，结合上述总结的基本概念和要点，具体讲解 TensorFlow 的使用。需要说明的是，考虑到本书篇幅有限，我们只会从下述代码中挑出一些重要的部分加以讲解。因此，讲解的示例代码并非是可以完整编译运行的代码。

本节完整的代码请访问 GitHub 查阅。[1]

3.3.1 定义计算图：训练语料库预处理

如前所述，我们的第一步就是定义从原始样本输入到高维特征向量输出的计算流程。TensorFlow 会根据我们定义的计算流程，自动生成相应的计算图。

我们需要将未标注的训练语料库进行一些预处理，并转换为特定格式的训练样本数据。在根据语料库构建输入样本之前，我们首先需要建立词典，并给予每一个词一个独一无二的索引值。这样，我们可以将词转换为对应的索引值，也可以根据索引值得到具体的词，具体代码如下。

```
  with zipfile.ZipFile(filename) as f:
    vocabulary = tf. compat.as_str(f.read(f.namelist()[0])).split()
  vocabulary_size = 100000
  data, count, dictionary, reverse_dictionary = build_dataset(vocabulary,
vocabulary_size)
  def build_dataset(words, n_words):
    """Process raw inputs into a dataset."""
    count = [['UNK', -1]]
    count.extend(collections.Counter(words).most_common(n_words - 1))
    dictionary = dict()
    for word, _ in count:
      dictionary[word] = len(dictionary)
    data = list()
    unk_count = 0
    for word in words:
      index = dictionary.get(word, 0)
```

1 https://github.com/tensorflow/tensorflow/blob/r1.9/tensorflow/examples/tutorials/word2vec/ word2vec_basic.py

```
        if index == 0:  # dictionary['UNK']
            unk_count += 1
        data.append(index)
    count[0][1] = unk_count
    reversed_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return data, count, dictionary, reversed_dictionary
```

其中，vocabulary_size 是一个可配置的参数，用来指定词典中所含的词的个数。函数 build_dataset 以语料库为输入，输出 data，count，dictionary 以及 reverse_dictionary 这 4 个全局变量。其中，data 是将原语料库中每一个词映射为其对应索引值得到的索引值序列；count 包含了语料库中词频最高的 vocabulary_size 个词，以及各自出现的次数；dictionary 的 key 是词，value 是词的索引值，取值范围为{0,1,…, vocabulary_size-1}，按照词频降序索引（0 代表 dictionary 里不包含的词）；reversed_dictionary 的 key 是索引值，value 是词本身。

【知识点讲解】需要指出的是，我们在构建词典时，往往只会把训练语料库中出现次数大于一定阈值（例如 5）的词放进词典中，然后进一步利用 Word2Vec 模型学习其分布式嵌入向量。而对于出现次数太少的词，我们往往将其视为生僻词，统一用“UNK”表示。这样做有几个好处：第一，如果把所有的词都放进词典中进行索引，则往往使得词典的索引值变得非常大。注意，词典中索引的词的个数，也等同于在 Skip-Gram 算法中输入向量（one-hot-vector）的维数，而过高的维数会降低训练算法的性能。第二，对于出现次数很少的词，它对应的训练样本也会很少。过少的训练样本往往无法得到其准确的分布式嵌入向量。基于这两点理由，我们只挑选词频最高的 vocabulary_size 个词进行索引，构建词典。事实上，对于绝大多数文本学习算法，都应用了类似的做法。

构建好了词典，并将训练语料库中的词用索引表示之后，我们进一步通过如下函数训练语料库中的句子，根据 Skip-Gram 的算法流程，转换为训练样本，即包含词及其上下文的词对（具体流程参见 3.2 节的例子）。其代码如下。

```
def generate_batch(batch_size, num_skips, skip_window):
  global data_index
  assert batch_size % num_skips == 0
  assert num_skips <= 2 * skip_window
  batch = np.ndarray(shape=(batch_size), dtype=np.int32)
  labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
  span = 2 * skip_window + 1  # [ skip_window target skip_window ]
  buffer = collections.deque(maxlen=span)  # pylint: disable=redefined-builtin
  if data_index + span > len(data):
    data_index = 0
  buffer.extend(data[data_index:data_index + span])
  data_index += span
  for i in range(batch_size // num_skips):
    context_words = [w for w in range(span) if w != skip_window]
    words_to_use = random.sample(context_words, num_skips)
    for j, context_word in enumerate(words_to_use):
      batch[i * num_skips + j] = buffer[skip_window]
      labels[i * num_skips + j, 0] = buffer[context_word]
    if data_index == len(data):
      buffer.extend(data[0:span])
      data_index = span
    else:
      buffer.append(data[data_index])
      data_index += 1
  # Backtrack a little bit to avoid skipping words in the end of a batch
  data_index = (data_index + len(data) - span) % len(data)
  return batch, labels
```

其中，batch_size 是一个可配置的参数，用来指定在一次模型权重的更新过程中，用多少条训练样本进行计算，也即 minibatch 中含有的样本个数。skip_window 是指上下文窗口的大小，一个词的上下文包括它前面以及后面最邻近的 skip_window 个词。num_skips 是指在当前词的上下文中，随机选出 num_skips 个词，以每个选出的词作为输出，当前词作为输入，构成一条训练样本。

【知识点讲解】minibatch 是在运用梯度下降法对模型参数进行优化时所涉及的一个概念。在利用梯度下降法更新模型参数的某一次迭代中，如果我们将所有的训练样本都用来计算梯度，则被称为批量梯度下降法（batch gradient descent）；如果我们每次更新只用一条训练样本来计算梯度，则被称为随机梯度

下降法（stochastic gradient descent）；如果我们每次更新用多个（但非全部）训练样本来计算梯度，则被称为微批量梯度下降法（minibatch gradient descent）。在微批量梯度下降法中，每次计算梯度时使用的训练样本个数就是 minibatch 的大小，也就是上述代码中的变量 batch-size。这是一个可配置的参数。如果 batch size 过小，则往往使得在每一次迭代中，梯度的估计误差较大；相反，如果 batch-size 过大，虽然其梯度的估计误差会比较小，但这会使得每一次迭代更新时所花费的计算量过大，降低了优化效率。因此，在实际过程中，batch-size 需要根据实际情况选择一个适中的值（例如，值为 8~1024）。此外，考虑到计算机运算单元的特点，batch-size 往往为 2 的整数幂，以较好地利用计算机的计算资源。

3.3.2 模型计算图的实现

首先，我们定义整个模型的输入。在 TensorFlow 中，一般使用 tf.placeholder 来定义输入。需要注意的是，当我们在定义计算图的时候，用 tf.placeholder 来定义的输入变量并不会被赋值，只是相当于一个占位符。它告诉编译器在实际执行这个计算图的时候输入数据的维度和数据类型等信息。只有在程序实际运行的时候，才会对 placeholder 定义的输入进行赋值，输入实际的数据，如以下代码所示。

```
batch_size = 128
embedding_size = 128
skip_window = 1
num_skips = 2
num_sampled = 64
graph = tf.Graph()
with graph.as_default():
  # Input data.
  with tf.name_scope('inputs'):
    train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
```

在上述代码中，train_inputs 和 train_labels 表示训练样本及其对应的样本标注，其各自对应一个 placeholder。此外，我们还设置了一些全局参数，其中 batch_size、skip_window、num_skips 表示的意义已经在前文解释过了。而 embedding_size 是词分布式嵌入向量的维数，num_sampled 是在负样本采样中，负样本的采样个数，也就是公式（5）中的n。

然后，定义计算图。首先实现编码映射层，其计算流程如公式（1）所示，具体实现代码如下。其中变量 embeddings 对应着公式（1）中的编码矩阵$\boldsymbol{W_1}$，它的每一个元素被初始化为一个取值为[−1.0,1.0]的随机值。embed 是输入样本经过编码映射之后得到的分布式嵌入向量，对应公式（1）中的$\boldsymbol{e_t}$。

```
# Look up embeddings for inputs.
with tf.name_scope('embeddings'):
  embeddings = tf.Variable(
      tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
  embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

之后，进一步定义解码映射层如下。其中 nce_weights 以及 nce_biases 是公式（2）中的 $\boldsymbol{W}_2$和 $\boldsymbol{b}_2$。

```
# Construct the variables for the NCE loss
with tf.name_scope('weights'):
  nce__weights = tf.Variable(
      tf.truncated_normal(
          [vocabulary_size, embedding_size],
          stddev=1.0 / math.sqrt(embedding_size)))
with tf.name_scope('biases'):
  nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
```

下一步就是计算公式（5）所示的代价函数，并采用微批量梯度下降法对模型参数进行优化，具体实现代码如下。

```
# Compute the average loss for the batch
with tf.name_scope('loss'):
   loss = tf.reduce_mean( tf.nn.nce_loss(
          weights=nce_weights,
          biases=nce_biases,
          labels=train_labels,
```

```
            inputs=embed,
            num_sampled=num_sampled,
            num_classes=vocabulary_size))

  # Construct the SGD optimizer using a learning rate of 1.0.
  with tf.name_scope('optimizer'):
    optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(loss)
```

需要说明的是，我们不需要关注优化算法实现的具体细节，TensorFlow 提供了内置的优化算法接口供我们调用，例如示例代码中的 tf.train.GradientDescentOptimizer。我们只需要指定它的一些参数即可，例如 learning rate，以及需要优化的目标变量 loss。

到现在为止，我们已经通过上述代码定义好了计算图及相应的代价函数与优化算法。下一步就是具体执行计算图，不断地读入微批量训练数据，根据定义的代价函数及优化算法、更新模型参数，直到得到最终的模型参数。在如下代码中，我们通过 feed_dict 将训练数据的 minibatch 当作输出，输送给前述定义好的计算图，并调用 session.run 执行运算图，以及更新模型权重。经过 num_steps 后，完成训练过程。

```
  # Add variable initializer.
  init = tf.global_variables_initializer()
  num_steps = 100001
  with tf.Session(graph=graph) as session:
    # We must initialize all variables before we use them.
    init.run()
    average_loss = 0
    for step in xrange(num_steps):
      batch_inputs, batch_labels = generate_batch(batch_size, num_skips,
skip_window)
      feed_dict = {train_inputs: batch_inputs, train_labels: batch_labels}
      # Define metadata variable.
      run_metadata = tf.RunMetadata()
      # We perform one update step by evaluating the optimizer op (including it
      # in the list of returned values for session.run()
  _, summary, loss_val = session.run([optimizer, merged, loss],
      feed_dict=feed_dict, run_metadata=run_metadata)
      average_loss += loss_val
```

3.4 Word2Vec 模型的局限及改进

Word2Vec 模型在对词的建模方面也有一些缺陷，在最近几年也陆续进行了一些改进，总结如下。[1]

- Word2Vec 模型无法很好地表达一些特定的短语。这些短语由多个词构成，但其语义却不是所含各个词的语义的简单叠加，因此，将这些短语所含词对应的分布式嵌入向量求和或求平均值得到的向量，其对应的语义和短语本身的语义相差很大。举例来说，“Best Buy”是美国的一家电子设备零售商，其语义表达了一个特定的公司名称。但是其组成的词汇“Best”和“Buy”的语义经过简单叠加或组合，完全无法表达“Best Buy”所对应的语义。在参考文献[1] 中，针对这个问题，采用了一个较为直接的做法，即先根据词共同出现的特性，检测出那些具有固定搭配的短语，然后将这些短语作为一个实体（可看作一个词）学习其分布式嵌入向量。当然这种做法的精度受限于短语的检测准确率。在参考文献[2]中，作者直接对 N-Gram 模型进行分布式嵌入向量的学习，也是考虑到短语建模的问题。
- Word2Vec 基于词的上下文（当前词临近的一个小窗口内的词）信息学习词汇语义的特点，使得它可能不足以表达一个词的完整语义，进而使得其学习得到的分布式嵌入向量存在一定的局限性。例如，“good”和“bad”都是形容词，其上下文很多时候也比较类似，例如“It was a good weather yesterday”和“It was a bad weather yesterday”都是非常合理也频繁出现的自然语言。这样，由于“good”和“bad”时常具有类似的上下文，使得它们在 Word2Vec 模型下学习得到的分布式嵌入向量也会比较相近。这样的结果从某方面看是合理的，因为相近的分布式嵌入体现了词的相似性（“good”和“bad”都是形容词，应用的语义场景类似），并且这种相

1 YOUNG T, HAZARIKA D, PORIA S, et al. Recent Trends in Deep Learning Based Natural Language Processing. arXiv: 1708.02709v8, 2018.

似性对某些应用是非常合理的，例如 POS Tagging。但是，我们也应该看到，“good”和“bad”表达了完全不同的情感取向，但它们对应的分布式表达却比较接近，这对于诸如情感分析（sentiment analysis）这样的应用来说就是一个大问题。

- 由于 Word2Vec 模型对每一个词只学习一个分布式向量表达，因此它无法很好处理一词多义的问题。在一篇论文中[1]，作者利用多语言之间翻译的对应信息帮助解决一词多义的问题。当然，这种方法也有很大的局限性。一词多义仍是自然语言处理中一个经典的难题。
- 由于 Word2Vec 模型是利用词出现时其上下文信息学习其分布式嵌入向量。因此，当一个词在语料库中出现次数比较少时，其对应的训练样本也就会比较少，因此，Word2Vec 对于这样的生僻词学到的分布式嵌入向量往往比较不准确。基于 Letter-Gram 的方法可以较为有效地对词根进行建模[2]，在一定程度上缓解了这种问题。在这种方法下，对于那些少见的生僻词，如果它含有一些特定词根，且其语义与词根有较好的联系，则其学到的分布式嵌入向量比基于传统的 Word2Vec 要好。

3.5 本章小结

本章介绍了被广泛使用的 Word2Vec 模型，包括其应用背景，具体的 Skip-Gram 模型及负样本采样方法，以及如何用 TensorFlow 实现这一算法。在应用深度学习处理文本和自然语言相关的问题时，利用 Word2Vec 模型将词转换为高维的分布式嵌入向量，往往是处理的第一步；而后，词的分布式嵌入向量

1 UPADHYAY S, CHANG K W, TADDY M, et al. Beyond bilingual: multi-sense word embedding using multilingual context. arXiv preprint arXiv:1706.08160, 2017.

2 BOJANOWSKI P, GRAVE E, JOULIN A, et al. Enriching word vectors with subword information. Transaction of the Association for Computational Linguistics 2017[J], 135-146.

作为后续深度神经网络模型的输入进一步学习得到我们想要的目标。利用 Word2Vec 模型学习词的分布式嵌入向量，无须预先对样本进行标注，可在大规模语料库中进行学习，其学习得到的分布式嵌入向量可以很好地刻画词的句法信息及语义信息。

此外，让我们再回顾一下 TensorFlow 在使用方面的一些要点。

- 计算图的概念对 TensorFlow 至关重要。它把所有的计算、操作、数据，通通抽象为计算图的一部分。当我们利用 TensorFlow 训练一个模型时，往往需要先定义模型所对应的计算图。具体来说，就是定义如何从原始输入数据，通过一步步计算，得到最终的代价函数值。TensorFlow 提供了大量的预先定义好的函数，以帮助用户快捷地构建需要的计算流程。此外，我们还需要指定特定的优化算法。类似地，TensorFlow 也提供了一系列常用的优化算法。定义好了从输入数据到代价函数值的计算流程，以及指定好优化算法，计算图就基本定义好了。
- 定义好的计算图只是告诉系统从输入到输出的逻辑是什么，并未真正执行。只有当创建 Session 对象，并调用 session.run 时，系统才会真正运行计算图的相关部分。在 session.run 中，我们可以指定需要运行的计算图的某一个或某一些节点，并通过 feed_dict 将必要的输入数据输入计算图中，用以计算我们所指定的输出节点。

参考文献

[1] MIKOLOV T, SUTSKEVER I, CHEN K, et al. Distributed representations of words and phrases and their compositionality. Proceedings of the 26th International Conference on Neural Information Processing Systems 2013[C], 3111-3119.

[2] JOHNSON R, ZHANG T. Semi-supervised convolutional neural networks for text categorization via region embedding. Proceedings of the 28th International Conference on Neural Information Processing Systems 2015[C], 919-927.

Deep Learning Model and Application

第 4 章

卷积神经网络在图像分类中的应用

卷积神经网络（Convolutional Neural Network，CNN）是一类特殊的前馈神经网络。与普通的前馈神经网络相同的是，它也是由多层神经元组成的，神经元之间通过可学习的权重连接，每个神经元（除输入层外）接受多个输入，输入经过加权（权重由边表示）求和及非线性映射得到输出，这些输出可作为下一层神经元的输入。与普通的全连接前馈神经网络不同的是，其不同层的网络节点之间并非全连接，而是局部连接，即神经元节点只与前一层的某些神经元节点有连接。卷积神经网络的核心概念包括卷积核、pooling 等。由于局部连接以及卷积核参数共享等策略，使得卷积神经网络的自由参数相比相同规模的全连接前馈神经网络大大减小，这在很大程度上提高了模型的泛化能力。由于卷积神经网络自身的特点，使得它非常善于处理图像或语音等自然连续信号，因此它最初主要应用于图像、视觉、语音等相关的领域。最近，它也被广泛地应用于自然语言处理相关的领域中，包括信息检索、句法分析、文档概要、实体检测等。

下面简要介绍一下卷积神经网络的历史。卷积神经网络的研究起源于对图像的处理。在图像领域，普通的全连接神经网络往往无法达到好的性能。例如，对一幅并不大的图像（128 像素×128 像素×3，即长宽各为 128 像素的彩色图像）来说，假设第一个隐藏层有 100 个节点，那么对全连接神经网络来说，从输入层到第一个隐藏层就会有 4915200（128×128×3×100）条边。而且，对于更大的图像及更大规模的隐藏层，网络的参数会成倍增加。这样数量巨大的自由参数，使得全连接前馈神经网络很容易过拟合，而无法得到好的泛化性能。

针对图像的特点，卷积神经网络被提出并得到成功应用。

LeNet5 是最早的卷积神经网络之一，被用于处理字符图像识别，它的提出大大促进了深度神经网络的发展。LeNet5 所采用的整体架构如图 4.1 所示[1]，目前大部分卷积神经网络仍然采用类似的架构。图 4.1 所示的卷积神经网络架构可以总结如下：

1 LECUN Y, BOTTOU L, BENGIO Y, et al. Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE 1998[C], 86(11) ,2278 - 2324.

- 在构建网络时，使用如下 3 种类型的隐藏层：卷积层、池化（pooling）层和全连接网络层。
- 卷积层通过卷积运算来提取局部特征。
- 池化层通过降采样提高特征的空间鲁棒性，并降低后续处理的运算复杂度。
- 网络中的非线性映射一般使用 tanh、sigmoid 或 ReLU 函数。
- 网络的最后一层（或几层）往往使用普通的全连接神经网络。

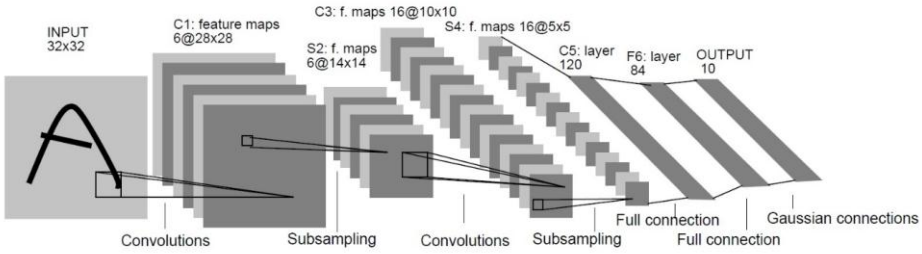


图 4.1　LeNet5 的网络架构

卷积神经网络的核心思想是：图像的像素在空间分布上有很强的相关性，图像可以由其局部特征的组合来表达；卷积运算是一种高效的提取图像局部特征的手段，不同的卷积核（卷积核是卷积运算的模板）可以提取不同的局部特征；在应用同一个卷积核时，我们用这个卷积核遍历整个图像以抽取图像中不同位置的局部特征；通过应用多个卷积核，并自动学习卷积核的权重，我们就可以实现图像特征的自动提取。由于卷积运算的自由参数是卷积核所包含的权重，而卷积核对应着局部运算，它的大小远远小于图像本身的尺寸，因此，它的自由参数相对于全连接神经网络会大大减少。

【知识点讲解】卷积运算是分析数学中的一种重要运算。以一维离散卷积运算为例，对于定义在整数域上的函数f以及卷积核g，卷积运算的定义如公式（1）所示。

$$y(n) = (f * g)(n) = \sum_{m=0}^{M-1} f(n-m)g(m) \quad (1)$$

其中，卷积核g的大小为M。从卷积的定义可以看到，对于任意一个指定点，卷积运算相当于用一个模板（即卷积核）与指定点附近的函数曲线进行点积。因此，如果在该指定点，卷积结果得到的值越大，则说明函数在该指定点所在区域的曲线与模板的曲线形状越接近。这也是卷积运算可以高效地提取局部特征的原因。

随后，出现了大量的卷积网络。它们大多继承了 LeNet5 的整体架构，但是又各自进行了各种各样的改进和变形。其中，AlexNet、Overfeat、VGG Networks、GoogleLeNet、ResNet、SqueezeNet 是一些非常著名的卷积网络。本章主要以 AlexNet 为例子来介绍卷积神经网络。

4.1 图像识别和图像分类的发展

图像识别和分类在实际中有非常广泛的应用，例如个人照片自动整理、社交网络中的身份识别、自动监控等。图像识别和分类也是机器学习领域里有着悠久研究历史的一个方向。在卷积神经网络被广泛应用之前，图像识别和分类都是基于人工定义的特征提取算法，然后在特征空间上进行分类器学习。那时，分类识别系统的性能在很大程度上取决于特征的设计。而要想设计出好的特征，则往往需要对目标领域有深入的理解，因此这种特征设计方法往往只适用于特定领域而难以推广到其他不同领域。例如在人脸检测中，广泛应用的是哈尔特征（Haar feature）；而在文字识别领域，广泛应用的是方向导数（directional derivative）特征。在深度神经网络兴起之前，特征工程（feature engineering）是传统机器学习算法中很重要的一部分。

近年来，深度学习技术蓬勃发展。从应用角度来看，深度学习一个显著的优势就是可以实现特征提取的自动学习，而不再需要人工定义特征提取算法。以卷

积神经网络为例，卷积核是进行局部特征提取的模板，它们是通过训练样本自动学习而得到的，并非人工预先指定的。自动学习特征提取算法大大提高了图像分类器设计的效率和性能。此外，图像、视频等数据也正在呈现指数增长，获取相应数据要比以往容易得多。当然，越来越强的计算能力及大规模分布式训练平台的日趋成熟，也是深度卷积神经网络在图像识别和分类领域得到广泛成功应用的前提和推动力。

对搜索与广告系统来说，图像分类也是一个非常重要的课题。例如，Google 和 Bing 等搜索引擎都提供了一些针对日常用品的广告，例如鞋、衣服等。广告主除提供产品的文字描述信息外，往往还提供产品的图片。这些图片信息可以和产品的文字描述信息一起作为源数据用于相应产品的建模。

4.2 AlexNet

AlexNet 是 Alex Krizhevsky 等人于 2012 年发布的用于图像分类的大规模深度卷积神经网络。它在 ImageNet 的子集 LSVRC-2010 上进行训练，训练样本有 120 万幅高分辨率图像，1000 个类别。在测试集上，它达到了当时最好的分类性能：第一候选分类结果的错误率为 37.5%；前五候选分类结果的错误率为 17.0%。此外，在 LSVRC-2012 竞赛集上，它的前五候选分类结果的错误率为 15.3%，远优于当时参加竞赛的亚军算法（前五候选分类错误率为 26.2%）。

AlexNet 之所以能够有优秀的分类性能，主要是因为它采取了以下技术：

- 使用了 ReLU 作为非线性激活函数，在一定程度上减少了深度网络中梯度消失的问题。
- 使用了 GPU（NVIDIA GTX580），并针对 GPU 实现了高度优化的卷积运算。由于提高了训练速度和效率，使得 AlexNet 可以使用更深的网络（5 个卷积层，3 个全连接层）以提高模型的表达能力。
- 使用局部响应归一化技术，模拟仿生学引入邻近活跃的神经元之间的竞争

机制。

- 使用存在重叠的池化感知域，而不是传统的非重叠的池化感知域。这使得池化变得更具鲁棒性。此外，AlexNet 还使用最大池化（max pooling）代替平均池化（average pooling），这样可以更好地提取显著特征，使得显著特征不会被平均池化弱化。
- 使用了 Dropout 技术，在一定程度上减少了过拟合（overfitting）。

AlexNet 的网络架构如图 4.2 所示。[1]它是一个 8 层网络：5 个卷积层和 3 个全连接层。下面介绍一下 AlexNet 网络的具体细节。

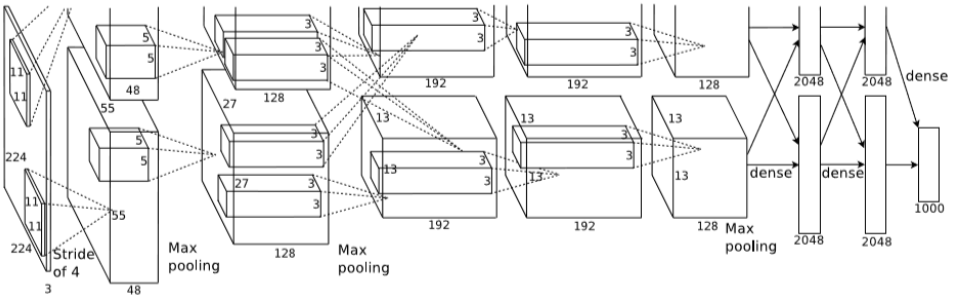


图 4.2　AlexNet 的网络架构

4.2.1　网络模型结构

1. 输入层

对于图像分类，其输入是一幅图像，用一个三维数组来表示。这 3 个维度分别对应着输入图像的长度、宽度和深度。如果输入图像是彩色图像，那么它的深度一般等于 3，对应着彩色图像的 RGB 这 3 个颜色通道；如果输入图像是灰度图，那么它的深度等于 1，相当于退化为二维数组。例如，对于一幅长度为 64 像素，宽度为 48 像素的彩色图像，其对应的输入数组大小为[64,48,3]。

1 KRIZHEVSKY A, SUTSKEVER I, HINTON G. ImageNet Classification with Deep Convolutional Neural Networks. Communications of the ACM 2017[J], 84-90.

事实上，对卷积神经网络来说，每一层神经元的输入（也即上一层网络的输出）均可表示为三维数组，且这 3 个维度依然对应着长度、宽度和深度。其他层与输入层（深度对应着不同的颜色通道）不同的是，其深度对应着不同的特征映射（feature）图。特征映射图的具体细节在随后会有介绍。

2. 卷积层及池化层

卷积层融合了 3 种架构设计思想以保证特征提取的鲁棒性，即针对一定程度的位移、缩放及形变的不变性。这 3 种架构设计思想是：感受野局部化（local receptive field）、权重模板共享化（shared weights）和池化（pooling）机制。

具体来说，感受视野局部化是指，卷积网络强迫每一个神经元只能和相邻层网络对应位置及其邻近小区域内的神经元产生连接；而与其他神经元无法连接。这样的架构使得卷积层学习得到的特征模板（一个特征模板也就对应着一个卷积滤波器）可以对输入信号的局部空间模式产生很强的反应。这种卷积层可以堆叠在一起形成深度卷积网络。随着堆叠层数的增加，其识别的局部模式的空间延拓范围会越来越大，也就越来越倾向于体现全局特征。因此，这种深度卷积网络是先学习和表达范围较小的局部特征，然后通过不断堆叠，将局部特征不断组合，实现对大范围的全局特征的学习和表达。

权重模板共享化是指，对于每一个特征模板（即每一个卷积滤波器），我们都用它遍历整个输入图像以提取相应的特征，而特征提取结果被称为一个特征映射图。这相当于对于图像的不同位置，会共享同一个特征提取模板（即卷积滤波器的权重参数），即特征提取与位置无关。这种设计是基于如下经验：

如果某种特征对于表达图像的某一个部分很有帮助，那么这个特征往往对于表达图像的其他部分也会有帮助。对于一个完整的卷积层，我们并非只使用一种特征模板进行特征提取，而是使用很多不同的特征模板进行特征提取，因此也会输出多个不同的特征映射图。因此，对于图像的每一个位置，我们都可以提取多种特征。另外一点需要说明的是，前一层卷积网络输出的特征映射图又可以被看

作是一幅图像(只是其深度不再对应颜色通道个数，而是对应特征映射图的个数，也即特征模板个数)，作为下一个卷积层的输入。多个卷积层可以堆叠在一起，形成深度卷积神经网络。

池化 (pooling) 这个机制是指对卷积层生成的特征映射图进行降采样。降采样的常见方式有最大池化 (max pooling)、平均池化 (average pooling) 等。池化机制主要有两个好处，一个是能够通过降采样来减小特征映射图的尺寸，降低后续操作 (如下一层卷积网络) 的计算复杂度。另外一个是可以提高特征提取的鲁棒性。例如，假设图像存在微小位移，如果位移的幅度小于池化窗口的大小，由于最大池化是从一个池化窗口中取特征映射图的最大值，那么这样使得最大池化后得到的特征映射图受图像位移的影响会很小。综上所述，通过运用感受野局部化(local receptive field)、权重模板共享化(shared weights)和池化(pooling) 这 3 种机制，卷积层可以很好地处理高维图像输入，并对图像的位移和形变有较好的鲁棒性。

以图 4.1 为例，其中输入层是一个长度、宽度、深度分别为 32 像素、32 像素、1 像素的灰度图像。第一个隐藏层是有 6 个特征模板的卷积层，相应地，它的输出含有 6 个特征映射图，对应 6 种特征提取方式。对于一个特征映射图 (也即一种特征提取方式)，它以一个神经元扫描整个输入图像；在输入图像的每一个位置，这个神经元仅接受输入图像中以当前位置为中心，一个大小为 5 像素×5 像素的小区域作为其输入，这个作为其输入的小区域被称为感知域；而神经元的输出，就是这个特征映射图在对应位置的取值。这样，这个特征映射图的神经元在扫描图像的每一个位置时，均有 25 (5×5) 个输入值，因此，就有 25 个可训练的连接权重，以及一个可训练的偏置，共 26 个可训练参数。而这 26 个参数定义了特征提取方式，对应着一种特征提取模板。注意，对于同一个特征映射图，它是使用同样的神经元扫描整个图像，因此其在图像的各个位置共享特征提取参数。也就是说，一个特征映射图只含有 26 个可训练参数。对图 4.1 中所示的第

一个隐藏层来说，它有 6 个特征映射图，因此共有 156（26×6）个可训练参数。

如前所述，每一个特征映射图对应着一种特征提取方式；特征映射图上某一位置的值是由以输入图像对应位置为中心的感知域作为输入，经由此特征映射图对应的特征提取后得到的输出值。如果特征映射图上的某个位置的值很大，则说明在此位置上有相应的特征被检测出来。一旦特征被检测出来，其精确的位置信息就没有那么重要了，而其大概的位置信息以及其与其他特征的相对位置会比较重要。举例来说，一旦我们知道输入图像在左上部区域含有一条大体水平的线段，在右上部区域含有拐角，在右部含有一条大致垂直的线段，那么我们可以大体识别出输入图像是数字“7”。在识别的过程中，特征的精确位置不仅不太重要，甚至对识别的泛化性能可能是不利的，这是因为数字“7”在输入图像中的位置可能会有各种不同的差异。

我们可以使用一种简单的做法来降低特征映射图中对特征精确位置信息的编码精度，这就是降低特征映射图的空间分辨率，可以通过池化层来实现。具体来讲，池化层是对原特征映射图进行了空间降采样。举例来说，平均池化是对原特征映射图进行局部平均，并在空间上进行降采样，用以降低特征映射图的空间分辨率，从而使得池化后得到的特征映射图对输入图像的位移和形变不那么敏感。仍以图 4.1 为例，图中第二个隐藏层就是一个池化层。这一层也含有 6 个池化网络，每一个池化网络都以一个卷积层输出的特征映射图作为输入。具体来说，对于一个特征映射图，我们将其划分为不重叠的 2×2 的小区域，每一个小区域可以被看作是池化网络神经元的感知域。在每一个 2×2 的小区域内，先计算区域内 4 个元素的平均值，然后乘以一个放大系数，再加上一个偏置系数，接着将结果输入 sigmoid 函数，最终得到池化层在当前位置的输出值。经过这样的处理，对于输入大小为 28 像素×28 像素的特征映射图，池化网络会输出一个大小为 14 像素×14 像素的池化结果图。由于池化层共有 6 个池化网络，因此输出 6 个池化结果图。需要说明的是，同一个池化网络具有相同的放大系数和偏置系数，并且这

个放大系数和偏置系数是在训练中通过学习得到的。放大系数和偏置系数可以用来控制 sigmoid 函数的非线性效果。如果系数比较小，那么 sigmoid 函数工作在准线性区域中，那么池化网络可以被理解为对原特征映射图进行平滑。如果系数比较大，sigmoid 函数工作在非线性区域；根据偏置的不同，池化网络可以被近似理解为“或操作”或“与操作”。

前面以图 4.1 中所示的第一个隐藏层和第二个隐藏层为例介绍了卷积层和池化层。事实上，我们可以将多个卷积层和池化层堆叠在一起，构成深度卷积网络。随着深度的增加，由于池化层的作用，后续卷积层的空间分辨率越来越低，特征映射图越来越小，其表达的特征越来越趋于全局化；与此同时，由于全局特征相对于局部特征更加多样，更加复杂，因此，后续层所包含的特征映射图往往越来越多，用以更好地表达全局特征。请注意，卷积层的神经元的输入是三维数组 [长度，宽度，深度]，其中深度可以用来表示前一层输出的特征映射图的序号。因此，后续卷积层可以用前一层网络输出的多个特征映射图作为输入。

由于网络的参数（包括特征提取模板系数）都是通过反向传播算法训练自动学习得到的，因此，卷积神经网络可以被看成是自动学习得到特征提取方法。一个卷积神经网络共享相同的特征提取模板，会使得自由参数的数量大大降低，提高了模型的泛化性能。例如，图 4.1 所示的网络共含有 340 908 个连接，但是只有 60 000 个可训练的自由参数，这正是因为使用了参数共享的策略。当然，参数共享也不可避免地在一定程度上限制了模型容量（model capacity）。但对图像来说，降低自由参数，提高泛化性能才是最关键的问题，因此，卷积神经网络在图像领域中得到了非常广泛且成功的应用。此外，通过增加特征映射图的个数及增加网络深度等方法，可以设计出适当的网络，使其具有足够的模型容量来对目标图像进行合适的处理和学习。

【知识点讲解】模型容量（model capacity）通俗地说是指模型能够学习到的模式的复杂程度，反映了模型的表达能力。例如，平面上的一条曲线是不可能由

仅含两个自由参数的线性模型所表达的。在这种情况下，线性模型容量就不足以刻画和表达我们的目标模式。一般来说，增加模型的复杂度，可以提高模型容量。例如，分段线性模型比线性模型有更多的自由参数，模型更复杂，也就有更高的表达能力。曲线可以由分段线性模型更好地近似和表达。对神经网络来说，更多的隐藏层，每层更多的节点，都增加了模型的复杂度，也同样增加了模型的表达能力。VC 维是模型容量的严格数学定义。[1]需要注意的是，在实际应用中，并不是模型的表达能力越强越好。虽然模型的表达能力越强，它越能学习更复杂的模式。但另一方面，它也越可能出现过拟合，使得模型的泛化能力变差。因此，模型容量需要和所研究的目标问题的复杂程度相匹配。

4.2.2 AlexNet 的具体改进

相比最传统的卷积神经网络（如 LeNet5），AlexNet 有一些很有趣的改进。在 4.2 节开头，我们将 AlexNet 最主要的改进进行了总结。下面，我们介绍这些改进的具体细节。

1. 使用 ReLU 作为非线性激活函数

在很长一段时间内，存在一种未经严格论证但被普遍认可的观点，即非线性激活函数应该是处处平滑且可导的函数。因此，大家常常使用 sigmoid 或 tanh 函数作为非线性激活函数。sigmoid 和 tanh 函数的表达式如公式（2）和公式（3）所示。作为非线性激活函数，它们的优点是处处平滑且可导，并且可把输出范围限制在（0,1）或（-1,1）。但是 sigmoid 和 tanh 激活函数有一个重要的缺点，就是当输入值比较大或比较小时，它的输出值相对于输出值的梯度会趋向于 0（即出现“梯度消失”问题）。这一点可以通过 sigmoid 和 tanh 函数的导数表达式看出来。以 sigmoid 函数为例，它的导数如公式（4）所示，显而易见，当x很大或很小时，这个导数值都非常小。由于反向传播算法是根据梯度进行权重更新的，

1 VLADIMIR V. The nature of statistical learning theory. Springer 2000[B]

因此当梯度非常小的时候，反向传播算法的更新过程会变得非常缓慢，有时甚至无法进行。这一现象被称为激活函数的“饱和”现象，引起了“梯度消失”问题。

$$\text{sigmoid}(x) = \frac{1}{1+\mathrm{e}^{-x}} \tag{2}$$

$$\tanh(x) = \frac{\mathrm{e}^{x}-\mathrm{e}^{-x}}{\mathrm{e}^{x}+\mathrm{e}^{-x}} \tag{3}$$

$$\text{sigmoid}'(x) = \frac{1}{1+\mathrm{e}^{-x}} \cdot \frac{\mathrm{e}^{-x}}{1+\mathrm{e}^{-x}} = \text{sigmoid}(x)(1-\text{sigmoid}(x)) \tag{4}$$

针对这个问题，参考文献[1]提出了一种新的非线性激活函数，即修正线性激活函数 ReLU（Rectified Linear Units）。它的定义如公式（5）所示。由于此函数在激活状态下（即x>0），梯度永远为 1，因此不存在梯度饱和现象。需要说明的是，ReLU 函数也带来了另一个问题，即有可能存在很少被激活的“死结点”。但如果有足够多的 ReLU 神经元，即使只有一部分神经元处于激活状态，往往也能够有足够的表达能力。

$$\text{ReLU}(x) = \max(0, x) \tag{5}$$

AlexNet 首次将 ReLU 函数应用在图像分类的卷积神经网络中，并取得了很好的效果。具体来说，由于 ReLU 克服了传统激活函数 sigmoid 和 tanh 的饱和问题，因此其训练优化过程的效率大大提高。例如，在 CIFAR-10 数据集上训练误差达到同样的 25%，即采用 ReLU 网络的训练的轮数只需要采用 tanh 的网络的近 1/5。正是因为训练效率大幅度提升，才使得 AlexNet 能够在这样大规模的数据上训练一个大规模的网络。

2. 采用多 GPU 进行并行训练

通过更合理的方式进行并行训练，以提高训练速度，从而能够使用更多的训练数据，始终是提高深度神经网络精度的一条重要途径。在 AlexNet 中，作者尝试了使用两个 GPU 进行并行训练，这最终使得模型的首选错误率和前五选错误

率分别降低了 1.7%和 1.2%。当然，AlexNet 对于训练的并行程度并不高，仅是同时使用两个 GPU。但随后，越来越多的模型使用了大规模的 CPU/GPU 集群对超大规模的训练数据进行分布式并行训练，以达到越来越高的分类准确率。也有非常多的算法和技术被陆续提出，用以解决大规模分布式并行训练中遇到的各种问题。关于这方面的最新进展可参见相关资料，本章不再做具体介绍。[1]

3. 局部响应归一化

局部响应归一化大多用于避免非线性激活函数的“饱和”问题，即通过合适的归一化操作，使得神经元的输入信号不至于太大或太小。对 ReLU 这种激活函数而言，虽然它没有像 sigmoid 或 tanh 函数那样的“饱和”问题，但是 AlexNet 的作者发现，进行适当的局部响应归一化，仍然有助于提高模型的准确率。具体来说，AlexNet 使用了如下的局部归一化方法。其中，对某一个卷积层来说，$a_{x,y}^{i}$表示这个卷积层中第i个特征映射图在位置（x,y）的值，$b_{x,y}^{i}$表示经过局部响应归一化后的值。局部响应归一化公式中的求和项是遍历相邻的n个特征映射图中相同的位置（x,y），N是在这一个卷积层中所含特征映射图的个数，k、n、α、β均是预先指定的参数，在 AlexNet 中，它们分别取值 2、5、0.0001 和 0.75。特征映射图的顺序是在训练前已经决定好的，大多是采用随机排序得到的。这种局部响应归一化可以实现“侧向抑制”，这是一种在神经科学中被发现的神经元组织活动现象，体现了那些邻近的非常活跃的神经元之间的一种竞争机制。通过在某些卷积层上应用这种局部响应归一化处理，模型最终的首选错误率和前五选错误率分别降降低了 1.4%和 1.2%。

$$b_{x,y}^{i}=a_{x,y}^{i}/(k+\alpha\sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)}(a_{x,y}^{i})^{2})^{\beta} \tag{6}$$

1 TAL B N, HOEFLER T. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. arXiv:1802. 09941v2,2018.

4. 使用存在重叠的池化感知域

如前文所述，池化层实现对卷积层中特征映射图的降采样。在传统卷积网络的池化层中，池化感知域是不重叠的。仍以图 4.1 为例，图中第二个隐藏层是一个池化层。它将输入的特征映射图划分为不重叠的 2×2 的小区域，即池化感知域。由于池化感知域无重叠，因此对于输入大小为 28 像素×28 像素的特征映射图，池化网络会输出一个大小为 14×14 的池化结果图。

事实上，我们可以把池化层看作是一系列池化神经元彼此相隔s，每一个池化神经元的池化感知域是以其所在位置为中心的大小为$z \times z$的区域。在图 4.1 所示的例子中，$s=z=2$。这时，不同池化神经元的池化感知域不重叠。但是当$s<z$时，不同的池化神经元的池化感知域就存在重叠。在 AlexNet 中，作者使用$s=2$，$z=3$这种配置。相比于传统的设置（$s=z=2$），这使得首选错误率和前五选错误率分别降低了 0.4%和 0.3%。此外，AlexNet 使用了最大池化，即取池化神经元感知域内的最大值而不是平均值作为神经元输入。这样做能更好地提取显著特征，使得显著特征不会被平均池化弱化。

5. Dropout

集成多个不同模型是一种非常有效的降低错误率的方法。但是，对大规模的深度卷积网络来说，训练一个模型往往需要几天或十几天。训练多个模型再进行集成的策略似乎代价太大了。为了解决训练代价太大的问题，Hinton 等人提出了可以高效地融合多个深度网络的训练方法，即 Dropout 方法。每一次输入训练数据时，它会将隐藏层的神经元的输出按照一定的概率随机设置为 0，使得这些被“丢弃”的神经元不再参与网络的前向传播，同时也不进行反向传播的参数更新。这样，每次当训练数据被输入网络时，网络会随机采样一个子网络进行真正的前向传播及利用反向传播更新参数，而被“丢弃”的神经元对应的连接权重则保持不变。这种技术相当于实现了一种多模型（被随机采样的子网络）的集成。从另一个角度看，由于一个神经元不能再固定地依赖某个特别的神经元的输入，

因此这种技术也降低了神经元之间的强相关性。也就是说，这种技术迫使网络学习那些更具鲁棒性的特征，即和神经元随机子集比较相关的特征，而不是和某个特定神经元高相关的特征。而实验结果也证实了，使用 Dropout 技术后，网络的泛化性能得到了大大的提高。

4.2.3 代价函数

在这里使用的代价函数是最常用的多类别分类的代价函数，即 softmax 函数。softmax 函数已经在第 3 章中进行过介绍，这里不再赘述。

4.3 应用 TensorFlow 实现 AlexNet

本节会介绍如何用 TensorFlow 实现前面介绍的 AlexNet。TensorFlow 的基本介绍及用其实现机器学习模型时所遵循的基本步骤可参见 3.3 节。需要说明的是，考虑到篇幅限制，我们只会从如下代码中挑出一些重要的部分加以讲解，因此，这里的示例代码并非是可以完整编译运行的代码。[1]

4.3.1 读取训练图像集

下面定义从原始样本输入到高维特征向量输出的计算流程。TensorFlow 会根据我们定义的计算流程，自动生成相应的计算图。

现在有很多不同的图像数据库，例如 MNIST（手写数字图像数据库，共有 70000 幅图像，图像分辨率为 28 像素×28 像素，包含数字 0~9 共 10 类图像）、CIFAR-10（包含 10 类图像，有飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船及卡车，每类有 6 000 幅图像，图像分辨率为 32 像素×32 像素）、Caltech-256（含有 256 类图像，共有 30607 幅图像，每类最少含有 80 幅图像）、LSVRC-2010

1 可运行的完整代码参见链接：https://github.com/tensorflow/models/blob/master/research/slim/nets/alexnet.py。

等。这些标准的图像数据库常常被用来作为基准以比较各种不同算法的效果。

各个图像数据库可能会有不同的存储格式，需要用不同的代码进行读取。这些标准图像数据库的读取代码很容易在网上找到，这里不再赘述。

4.3.2 模型计算图的实现

模型计算图的实现代码如下所示。其中，inputs 是网络的输入，它是一个大小为[batch_size,height,width,channels]的四维矩阵。slim.conv2d 用来实现卷积网络层，slim. max_pool2d 用来实现池化层，slim.dropout 用来实现前述的 Dropout 机制。需要说明的是，在如下所示的 GitHub 上 AlexNet 的具体实现中，已经将原 AlexNet 论文中最后三层全连接网络替换为卷积网络。

```
with tf.variable_scope(scope, 'alexnet_v2', [inputs]) as sc:
  end_points_collection = sc.original_name_scope + '_end_points'
  # Collect outputs for conv2d, fully_connected and max_pool2d.
  with slim.arg_scope([slim.conv2d, slim.fully_connected, slim.max_pool2d],
                      outputs_collections=[end_points_collection]):
    net = slim.conv2d(inputs, 64, [11, 11], 4, padding='VALID',
                      scope='conv1')
    net = slim.max_pool2d(net, [3, 3], 2, scope='pool1')
    net = slim.conv2d(net, 192, [5, 5], scope='conv2')
    net = slim.max_pool2d(net, [3, 3], 2, scope='pool2')
    net = slim.conv2d(net, 384, [3, 3], scope='conv3')
    net = slim.conv2d(net, 384, [3, 3], scope='conv4')
    net = slim.conv2d(net, 256, [3, 3], scope='conv5')
    net = slim.max_pool2d(net, [3, 3], 2, scope='pool5')

    # Use conv2d instead of fully_connected layers.
    with slim.arg_scope([slim.conv2d],
                        weights_initializer=trunc_normal(0.005),
                        biases_initializer=tf.constant_initializer(0.1)):
      net = slim.conv2d(net, 4096, [5, 5], padding='VALID',
                        scope='fc6')
      net = slim.dropout(net, dropout_keep_prob, is_training=is_training,
                         scope='dropout6')
      net = slim.conv2d(net, 4096, [1, 1], scope='fc7')
      # Convert end_points_collection into a end_point dict.
      end_points = slim.utils.convert_collection_to_dict(
          end_points_collection)
      if global_pool:
```

```
        net = tf.reduce_mean(net, [1, 2], keep_dims=True, name='global_pool')
        end_points['global_pool'] = net
      if num_classes:
        net = slim.dropout(net, dropout_keep_prob, is_training=is_training,
                           scope='dropout7')
        net = slim.conv2d(net, num_classes, [1, 1],
                          activation_fn=None,
                          normalizer_fn=None,
                          biases_initializer=tf.zeros_initializer(),
                          scope='fc8')
        if spatial_squeeze:
          net = tf.squeeze(net, [1, 2], name='fc8/squeezed')
        end_points[sc.name + '/fc8'] = net
    return net, end_points
```

4.4 本章小结

本章以 LeNet5 及 AlexNet 为例，介绍了卷积神经网络的基本概念、结构、原理及其在图像分类中的应用。需要说明的是，从 LeNet5 及 AlexNet 被提出以后，又相继出现了各种各样的卷积神经网络。它们采用的整体架构有很大相似之处，但是也有各种各样的不同和创新。下面对其中一些著名的网络进行简要总结，供读者参考。

- ZFNet：它是 ILSVRC 2013 的冠军。它对 AlexNet 的架构中的一些系统参数进行了重新调整，包括卷积模板的大小、池化感知域的重叠参数等。细节请参看相关资料[1]。
- GoogLeNet：它是 ILSVRC 2014 的冠军。它最主要的贡献是设计了 Inception 模块。Inception 模块可以大大降低自由参数的个数，因此，可以使得网络使用更多的不同维度的卷积模板提取特征，从而提升模型性

1 ZEILER D M, FERGUS R. Visualizing and Understanding Convolutional Networks. arXiv:1311.2901. 2013.

能。具体细节请参看相关资料[1]。

- ResNet：它是 ILSVRC 2015 的冠军。它引入了跳跃连接（skip connection）机制，并非常广泛地使用局部响应归一化（batch normalization）策略。ResNet 是目前用于图像识别方面性能最好的卷积神经网络，其往往也被当作图像识别领域的工程实践中首选尝试的卷积神经网络。具体细节请参考看相关资料[2]。

参考文献

[1] NAIR V, HINTON G. Rectified linear units improve Restricted Boltzmann Machines. Proceedings of the 27th International Conference on International Conference on Machine Learning 2010[C], 807-814.

1 SZEGEDY C, LIU W, JIA Y, et al. Going deeper with convolutions. arXiv: 1409.4842v1,2014.

2 HE K, ZHANG X, REN S, et al. Deep Residual Learning for Image Recognition. arXiv: 1512.03385,2015.

Deep Learning Model and Application

第 5 章
递归神经网络

在普通的前馈神经网络中，每一个样本通常被表示为一个高维向量，并作为一个整体同时输入到神经网络中。在样本的不同维数据之间不存在时序的先后顺序问题。然而，对于具有时序关系的输入数据来说（例如自然语言文本、股票价格走势），神经网络应该考虑到不同数据之间的时序关系，才能够对目标问题进行合理的建模。递归神经网络（Recurrent Neural Network，RNN）就是能够处理时间序列数据的一类深度神经网络，其已经在工程实践中得到广泛应用。本章主要介绍递归神经网络的一些基本概念，关于具体的 RNN 模型及其在信息检索领域的应用会在后续章节中介绍。

5.1 递归神经网络应用背景介绍

递归神经网络最主要的应用场景是处理时间序列数据。常见的实际应用包括语言模型、文本生成、语音识别、机器翻译、问答系统、股票预测，等等。以语音识别为例，在 21 世纪初，基于隐马尔可夫模型（HMM）的算法主导着语音识别领域。随后，在 2003 年，基于长短期记忆模型（LSTM，是递归神经网络的一种）的算法已经取得了与当时最先进的基于 HMM 的主流系统相类似的性能。在 2007 年，LSTM 在关键词识别任务中已经超越主流的 HMM 算法。在 2013 年，LSTM 在著名的 TIMIT 音素识别数据集上达到了最好的性能。除了在语音识别领域的成功应用，LSTM 还在手写文字识别领域（手写文字也可以看作是序列数据）取得成功。例如，它在 ICDAR 2009 的手写文字识别（三种语言：法语、阿拉伯语、波斯语）竞赛中获得冠军。此外，LSTM 还与其他技术进行组合，以进一步提高系统的性能。例如，它作为算法的一部分被应用于蛋白质分析、手写文字识别、语种识别、韵律预测、语音合成等方面。[1]

1 SCHMIDHUBER J. Deep learning in neural networks: An overview. Neural Networks 2015[J] 61,85-117.

5.2 递归神经网络模型介绍

不同于前馈神经网络（神经元之间的连接不形成环路），在递归神经网络（RNN）中，神经元连接形成有向环，这种性质使得它可以处理序列数据。具体来说，RNN 依次以序列的每个输入数据及其对应的前一时刻系统状态作为输入，通过一定的函数映射，得到当前时刻的系统状态。这样，系统当前时刻的状态既与当前时刻的输入数据有关，也与前一时刻的系统状态有关，以此机制来刻画数据时序上的相关性。我们可以从另一个角度来理解 RNN，即它可以将过去发生的信息以一种方式进行编码存储下来（用系统状态来刻画），相当于拥有一定的记忆功能。理论上，递归神经网络可以利用任意长序列中的信息进行编码，但实际上，它往往仅能够较好地编码之前少数几步的信息（稍后将详细介绍）。

5.2.1 递归神经网络模型结构

最简单的递归神经网络如图 5.1 所示。图 5.1 左边的是递归神经网络的普通形式，它可以被展开为右边的类似前馈神经网络的形式。展开形式只是根据时间序列把反馈形式表达为普通的非反馈形式。例如，如果我们关心的序列是具有 10 个单词的句子，则网络将展开为 10 层神经网络，每个单词作为其中一个层的输入信号。图 5.1 中，x_t表示在时刻t的输入数据；s_t表示时刻t的系统隐藏状态变量，这一隐藏状态变量表达了系统对从起始时刻到当前时刻所有输入信息的一种编码，它是由系统前一时刻的状态s_{t-1}以及系统当前输入x_t共同计算得到的；y_t表示时刻t的系统输出。

递归神经网络的数学表达如公式（1）所示，其中函数σ表示神经网络的激活函数，常用的有 tanh 或者 ReLU。系统的初始状态S_0往往被置为 0。由公式（1）可以看出，RNN 中不同时刻的网络参数（W,U,V）都是共享的，这样做的目的是通过参数共享来大幅度降低自由参数的个数，避免过训练。同时，它反映了 RNN 在不同时刻进行的操作（函数映射）是相同的，不同的仅是系统当时的状

态及输入信号。

$$S_t = \sigma(WS_{t-1} + UX_t)$$
$$Y_t = \sigma(VS_t) \tag{1}$$

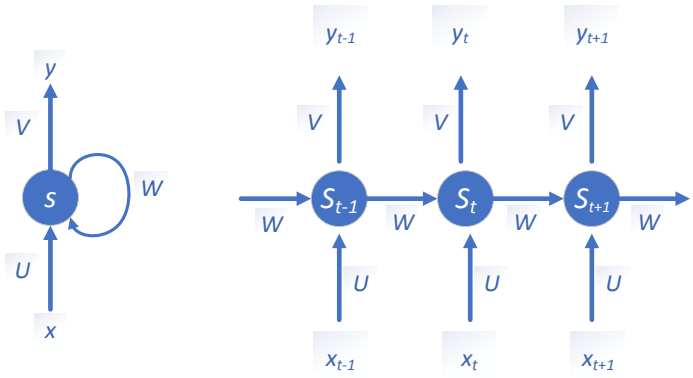


图 5.1　递归神经网络结构示意图

需要说明的是，图 5.1 仅是 RNN 的一个基本结构图。在实际应用中，网络的具体结构可以根据实际应用的需要进行各种各样的修改。下面，我们介绍两种常见的 RNN 的变种模型。

5.2.2　双向递归神经网络

不同于图 5.1 中从左向右的单向网络，我们还可以构建双向网络，以便于更好地利用双边的上下文信息，这就是所谓的双向递归神经网络（Bidirectional Recurrent Neural Network，BRNN）。[1]因为双向递归神经网络具有双向性，所以当它应用于时间序列数据时，信息不仅可以按照正常的时间序列传递（即过去的信息作为输入信息影响系统未来的状态），而且未来的信息也可以反向传递，为推测过去的系统状态提供信息。双向递归神经网络的数学表达如公式（2）所示，其中，角标 1 和 2 分别表示前向递归神经网络和后向递归神经网络的参数。

1 Schuster, Mike, Kuldip K. Paliwal, Bidirectional recurrent neural networks, IEEE Transactions on Signal Processing, 1997.

$$
\begin{aligned}
S_t^1 &= \sigma(W^1 S_{t-1} + U^1 X_t) \\
S_t^2 &= \sigma(W^2 S_{t+1} + U^2 X_t) \\
Y_t &= \sigma(V^1 S_t^1 + V^2 S_t^2)
\end{aligned}
\tag{2}
$$

在求解双向递归神经网络时，通常是将此双向递归神经网络视为两个递归神经网络的组合来应用反向传播算法。一旦根据反向传播算法计算出两个梯度，就会同时更新两个递归神经网络各自的权重。

5.2.3 长短期记忆模型

长短期记忆模型（Long Short-Term Memory，LSTM）由 Hochreiter 和 Schmidhuber 于 1997 年发明，是递归神经网络的一项重要突破。[1]引入 LSTM 是为了克服 RNN 不能有效建模长距离依存关系的问题。[2]LSTM 通过引入特别设计的结构和单元（记忆单元、门单元），有效地缓解了 RNN 中刻画长距离依存关系中常常面临的梯度消失问题。

LSTM 由以下一些关键部分组成。[3]

- 状态：它用于给系统的输出提供信息，具体可包含输入数据、隐藏状态、输入状态、内部状态。
  - 输入数据：用 x 表示。时间序列中每一时刻的数据，就是其对应时间步骤的输入数据。
  - 隐藏状态：前一时间步骤的隐藏状态，用h表示。它主要表达了前一时间步骤的网络状态，与传统递归神经网络中的隐藏状态的含义一样。
  - 输入状态：前一时间步骤的隐藏状态和当前输入数据的组合，用 i 表

1 HOCHREITER S, SCHMIDHUBER J, Long short-term memory. Neural computation[J], 9(8), 1997:1735-1780.

2 Yoshua Bengio, Patrice Simard, Paolo Frasconi, Learning Long-Term Dependencies with Gradient Descent is Difficult, IEEE Transactions on Neural Networks, Vol. 5, No. 2, 1994.

3 WANG H, RAJ B. On the Origin of Deep Learning. arXiv preprint arXiv:1702.07800v4, 2017.

示，具体如公式（3）所示。即将神经网络当前的输入数据与前一时间步骤神经网络的隐藏状态进行组合，共同构成当前时刻的神经网络输入。

$$i^t = \sigma(W_{ix}x^t + W_{ih}h^{t-1} + b_i) \quad (3)$$

— 内部状态：充当“记忆”的功能和角色，是神经网络基于所有历史数据进行编码后所表达的信息总和，用m表示。

- 控制门：控制门是 LSTM 很有特色也很关键的机制设计，它用于决定状态之间的信息流向。控制门的引入，较好地避免了传统 RNN 中的梯度消失问题（梯度消失是由于在多层网络中根据链式法则进行梯度连乘导致的）。

— 输入门：用于决定输入状态的信息是否会进入内部状态，用g表示，具体如公式（4）所示。如果输入门取值为 1，那么当前时刻的输入状态会完全进入内部状态，即神经网络的内部状态会记住当前时刻的输入状态，将其整合进当前的内部状态之中；反之，如果输入门取值为 0，那么当前时刻的输入状态会被忽略，相应地，神经网络当前的内部状态会保持前一时刻的内部状态。

$$g^t = \sigma(W_{gi}i^t) \quad (4)$$

— 遗忘门：它用于决定内部状态是否遗忘前一时刻的内部状态，用f表示，具体如公式（5）所示。如果遗忘门取值为 1，那么前一时刻的内部状态会被完全阻拦，相应地，神经网络的当前内部状态只会记住当前时刻的输入状态，而完全忽略前一时刻的内部状态；反之，如果遗忘门取值为 0，那么前一时刻的内部状态会完全进入当前的内部状态。

$$f^t = \sigma(W_{fi}i^t) \quad (5)$$

— 输出门：用于决定内部状态是否会将其值传送给网络的隐藏状态，用o表示，具体如公式（6）所示。这里需要解释的是，系统的隐藏状态可以看作是内部状态的一个子集。由于输入状态是由输入数据及隐藏状态组合而来的，因此输出门决定了系统内部状态中哪些部分用于和下一时刻的输入数据一起来形成下一时刻的神经网络输入状态。

$$o^t = \sigma(W_{oi} i^t) \tag{6}$$

综上所述，我们用两个公式来最终表达 LSTM，具体如公式（7）和公式（8）所示，其中⊙表示元素乘积（element-wise product）。从公式（7）中可以看出，神经网络的某一时刻的内部状态是当前时刻的输入状态和上一时刻的内部状态的加权求和，而输入门和遗忘门就是控制着加权的权重。考虑两个极端情况：当输入门为 1 且遗忘门也为 1 时，系统内部状态完全等于当前时刻的输入状态，而彻底遗忘前一时刻的内部状态；当输入门为 0 且遗忘门也为 0 时，系统内部状态完全等于前一时刻的内部状态，而彻底忽略当前时刻的输入状态。

$$m^t = g^t \odot i^t + （1 - f^t） \odot m^{t-1} \tag{7}$$

$$h^t = o^t \odot m^t \tag{8}$$

通过公式（3）至公式（8），我们完整地定义了 LSTM。其中涉及的所有权重（即W_{ix}、W_{ih}、W_{gi}、W_{fi}、W_{oi}）都是在训练期间需要学习的参数。通过引入控制门，LSTM 可以学会记住长距离的依存关系，也可以在必要时忘记过去，这使得它成为一个灵活而强大的模型。

自从 LSTM 被提出以来，人们一直在尝试改进它。例如，增加一个“peephole connection”，将内部状态作为控制门输入的一部分，因此使得控制门可以使用

内部状态所包含的信息。[1]或者引入门控递归循环单位 GRU，来简化 LSTM。[2]具体来说，它将内部状态和隐藏状态合并为同一个状态，并将遗忘门和输入门合并成为一个简单的更新门。此外，还有 Depth Gated RNN 等都是针对 LSTM 的改进工作。[3]尽管不断有新的 LSTM 的变体被提出，但 Klaus Gre 等进行了一项大规模的比较实验，其实验结果表明，所有 LSTM 的变体都不能显著地改进标准 LSTM。[4]这一结果提示，也许修改 LSTM 的单元内部结构不是一个好的方向，而注意力模型则是一个更有希望的方向。我们将在第 8 章具体介绍含有注意力机制的 LSTM 及其在信息检索领域的具体应用。

5.3 递归神经网络展望

RNN 和 LSTM 除了被当作单独的模型被广泛应用，还与很多其他模型以各种方式进行组合加以应用。例如，将卷积操作直接构建到 LSTM 中，就得到了 ConvLSTM。[5]具体来说，就是将 LSTM 的单元中各个控制门所含的矩阵乘法运算用卷积操作来代替，用以提取和利用高维数据所具有的空间分布特征。此外，卷积神经网络（CNN）和 LSTM 经过简单组合，被称为 CNN-LSTM 模型。它

1 GERS A F, SCHMIDHUBER U. J. Recurrent nets that time and count. Neural Networks. Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks 2000[C] .

2 CHO K, VAN B, ENBOER M, et al. Learning phrase representations using rnn encoder-decoder for statistical machine translation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) [C], 2014.

3 YAO K, COHN T, VYLOMOVA K, et al. Depth-Gated Recurrent Neural Networks. arXiv preprint arXiv:1508.03790v 2,2015.

4 GREFF K, SRIVASTAVA K. R, KOUTNK J, et al. Lstm: A search space odyssey. IEEE Transactions on Neural Networks and Learning Systems 2017[J], 28(10), 2222-2232.

5 Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-kin Wong, Wang-chun Woo, Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting, arXiv preprint arXiv:1506.04214, 2015.

是先用 CNN 来处理数据，形成一个序列数据，然后再将此序列数据作为输入送给 LSTM，其典型应用包括图像序列分析（视频分析）。此外，LSTM 还可以和条件随机场组合在一起使用，用于序列标注。[1]

需要强调的是，由于传统激活函数的梯度值往往小于 1，因此当使用反向传播算法根据链式法则计算梯度更新的时候，误差信号会随着追溯的时间步长（对应着神经网络的深度）呈指数减小，因此长距离依存关系（即很多步以前的信号变化对当前信号的影响）将丢失。这种梯度消失问题是训练 RNN 最根本的问题。通过引入新的激活函数（例如 ReLU）以及新的网络结构（例如 LSTM）可以在一定程度上缓解梯度消失问题。但是，应该注意到，这些解决方案只是通过引入了巧妙的设计来绕过这个问题，而非从根本上解决这个问题。虽然这些方法在实际应用中很有效，但 RNN 的上述问题仍有待从根本上加以解决。

5.4 本章小结

本章内容是对递归神经网络的一个简要概述。我们介绍了递归神经网络的基本概念、典型的应用场景，以及一些著名的变体，如 BRNN、LSTM。此外，本章还对 RNN 存在的问题以及未来的研究方向进行了简要总结。在后续的章节中，我们会具体介绍几种 RNN 模型及其在实际产品中的应用。

参考文献

[1] LIPTON C. Z, BERKOWITZ J. A critical review of recurrent neural networks for sequence learning. arXiv:1506.00019v4,2015.

1 HUANG Z, XU W, YU K. Bidirectional LSTM-CRF Models for Sequence Tagging. arXiv preprint arXiv:1508.01991v1, 2015.

Deep Learning Model and Application

第 6 章

DeepIntent 模型在信息检索领域的应用

我们在第 5 章中已经介绍了 RNN 的基本结构和原理。本章将介绍一种含有注意力机制的 RNN 模型，以及其在信息检索领域的应用。

RNN 可以将一个序列作为输入，通过其网络运算将输入序列映射为一个状态向量（参见第 4 章）。这种做法的好处是可以处理任意长的序列。而其缺点是，无论输入序列有多长、包含多少信息量，最终都要被映射成唯一的一个状态变量（维数通常是几百维）。这意味着，输入序列越长，包含的信息越多，最终的状态向量就会丢失越多的信息。而含有注意力机制的 RNN 模型就是为了克服这一缺点而提出的。事实上，RNN 在依次处理输入序列的各个元素时，会对应产生一系列的中间状态向量，而这一系列的中间状态向量更好地保存了输入序列的信息。因此，不同于最基本的 RNN 模型直接把最后一个状态向量作为最终的表达向量，含有注意力机制的 RNN 模型综合考虑了所有生成的中间状态向量，通过一个额外的注意力网络（attention network）把这一系列中间状态向量进一步映射为最终的表达向量。在大部分的论文中，注意力机制体现为一个权重向量，表达了人类认知中类似注意力的概念，体现各个中间状态向量对最终表达的重要程度，因此得名注意力模型（attention model）。注意力模型与人类的直观认知相符。例如，同一序列中不同元素的重要程度的确会有较大区别；而人们往往会把注意力集中在局部的非常重要的部分上。比如一幅图像，人们往往会把注意力集中在图像的主体事物上，而忽略次要的背景信息。

注意力机制在深度学习领域获得了极大关注。含有注意力机制的 RNN 模型在自然语言处理的相关领域得到了广泛应用。下面，我们将结合一个实际的文本检索应用，以及一个具体的 DeepIntent 模型，来介绍如何应用含有注意力机制的 RNN 模型解决实际问题。

6.1 信息检索在搜索广告中的应用发展

在搜索广告领域，对用户查询的搜索意图及广告商的广告意图进行精准匹配

是至关重要的。只有做到用户查询和广告之间的精准匹配，才能提高用户的使用体验，同时给广告主带来正确的受众，从而增加广告主的营收，建立一个健康可持续发展的搜索广告市场。例如，用户在搜索引擎中输入“佩奇系列图书”，算法应当理解用户想要搜索图书，而图书的名称是有关小猪佩奇的。对此，标题为“小猪佩奇（全 10 册）”的广告即为正确匹配的相关广告。而标题为“【Petkit 佩奇】智能宠物循环活水机”的广告即为错误匹配的无关广告。

传统的匹配算法往往以 bag of words（词袋）作为用户查询或是广告内容的特征表达，并基于此计算相似度。各种各样的相似度度量被广泛研究过。例如，TF-IDF 加权或 BM25 等。bag of words 这一特征表达方式，每一维代表一个词，特征向量维数往往非常高（等同于词表中词的个数），且非常稀疏，这为后续的学习（例如相似度学习）带来很大困难。此外，对于 bag of words 这一特征表达方式，它把所有不同的词都表示为完全无关的不同维特征。即使是词义非常接近的近义词，也会被表示为完全无关的不同维特征，因此它不能很好地处理近义词的问题。可见，bag of words 这种特征表达方式不能有效地体现词的语义信息。反之，基于深度神经网络（DNN）的算法，是将词序列（例如短语或句子）通过神经网络映射为一个嵌入向量，此向量的维数往往远远低于词表中词的个数。较低的维数使得基于嵌入向量的表达非常紧凑，会大大减少后续学习过程中因过高维数引起的维数灾难问题，也有效避免了特征稀疏的。此外，通过适合的 DNN 学习，算法能更好地编码词及其上下文的语义信息，使得近义词往往有相似的嵌入向量。由于上述优点，基于 DNN 的嵌入向量在相关领域得到了广泛应用并取得很好的效果。

对于基于 DNN 的信息检索算法来说，将词序列（例如短语或句子）进行适合的映射以得到准确的表达向量，是实现精准匹配的基础。换句话说，意图相似的词序列，经过映射后，得到的表达向量也应当比较类似（距离较近）；而语义完全不同的词序列，经过映射后，得到的表达向量也应当非常不同（距离较远）。

这样的网络映射，才是合适有效的映射。在本章的后续内容中，我们会介绍如何设计合适的 DNN 结构，选择合适的代价函数和优化算法，来学习这种有效的映射。

现在我们以实际的搜索广告为例，介绍如何在广告系统中应用这一技术。事实上，在将用户查询和广告内容通过本章介绍的网络映射为最终的表达向量之后，我们就可以在表达向量空间计算每一对用户查询和广告内容的距离（本章使用余弦距离）。这样，对于任何一个用户查询，我们都可以选择出和它距离最近的广告内容作为相关广告的候选展示给用户。当然，在实际的广告系统中，利用本算法选出的候选广告还需要经过一系列的后续筛选（例如：选出的广告竞价不能太低，用户点击此候选广告的概率不能太小，此候选广告所对应的广告商的广告费用仍有余额）及竞价，获胜的几个最优候选广告才会最终展示给用户。

6.2 含有注意力机制的 RNN 模型

如前所述，本章内容是在 RNN 模型的基础上引入了注意力机制。在具体介绍算法之前，先举几个例子，以解释为什么要引入注意力机制，以及它有什么好处。

在利用深度神经网络学习映射函数时，我们需要注意到，不同的词对于表达词序列（用户查询或广告内容）整体意图的重要程度是非常不同的。例如，用户查询问题为“Microsoft office for Mac”，去掉“Microsoft”这个词不会对原有查询的主要意图产生影响，因此“Microsoft”这个词在“Microsoft office for Mac”这一词序列中并不重要。但是，在 “hotel near by Microsoft”这样的用户查询中，“Microsoft”对于意图的表达就非常重要。因此，下面将要介绍的算法，是在 RNN 模型的基础上，引入了注意力机制，用以刻画各个词对于词序列整体语义的重要程度，进而学习得到整体词序列的最终表达向量。

具体来讲，本算法中的注意力机制，作用类似于池化（pooling）。例如 max

pooling、mean pooling 或 last pooling，其作用都是将一系列的中间状态向量经过一定的运算，映射为一个最终的表达向量。它们的不同之处在于，max pooling 是取序列中的最大值为结果；mean pooling 是取序列中的均值作为结果；而注意力机制具有更好的动态自适应性，它根据序列中各元素的重要程度将其加权得到最终值。此外，last pooling 只以序列最后一个输出得到的状态向量作为最终的表达向量，而注意力机制利用了所有的中间状态向量并综合考虑了其重要程度，这使得带有注意力机制的 RNN 模型可以比 last pooling 更好地刻画长序列。

含有注意力机制的 RNN 模型的整体框架如图 6.1 所示。它包含两大部分：（1）两个 DNN 分别将用户查询和广告内容映射为两个具有相同维数的向量。这两个 DNN 具有相同的结构，但有不同的网络权值。我们将这个 DNN 称为编码器。（2）基于用户点击信息的代价函数，用以指导编码器网络权值的学习。即，通过优化算法调整编码器的网络权值，使得其对应的代价函数值最优化。下面，我们先介绍编码器网络的具体结构，再介绍代价函数的构造。

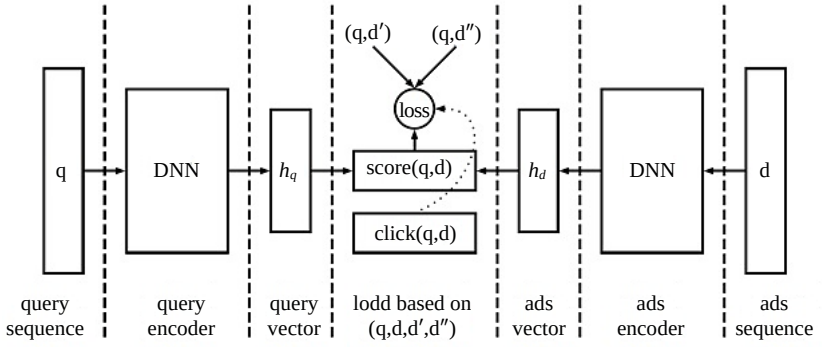


图 6.1　含有注意力机制的 RNN 模型的整体框架

6.2.1　网络模型结构

1. 输入层

编码器的输入是一个词序列，我们用$\boldsymbol{X}=\{\boldsymbol{x}_1,\boldsymbol{x}_2,\ldots,\boldsymbol{x}_t\}$来表示。其中，$t$是词

序列的长度。$\boldsymbol{x}_t \in \mathbf{R}^V$是第$t$个词的 one-hot-vector 表示，它是一个维数为V的向量，V是词典中所有不同词的个数。例如，在一个语料库中，共有 10 000 个不同的词，那么V=10 000；每个词对应一个固定的维数索引，并用一个 10 000 维的向量表示，且此向量只有目标词对应的那一维为 1，其余均为 0。这种表示即被称为 one-hot-vector。

2. 词嵌入层

在词嵌入层（word embedding layer），对于输入词序列的每个词$\boldsymbol{x}_t$，词嵌入层将其 one-hot-vector 通过如下的线性变换映射为一个低维空间的向量$\boldsymbol{e}^t$，称为嵌入向量。

$$\boldsymbol{e}_t = \boldsymbol{W}_{\text{emb}}\boldsymbol{x}_t, \text{ s.t. } \boldsymbol{W}_{\text{emb}} \in \mathbf{R}^{d_{\text{emb}} \times V} \tag{1}$$

其中，$\boldsymbol{W}_{\text{emb}}$的每一列对应着词表中相应词的嵌入向量表达。通过词嵌入层，每个词的表达由极高维（维数等于词表中词的个数，可多达百万量级）的 one-hot-vector，降维到紧致的文本表达空间（通常为数百维），这使得以词嵌入作为输入的后续模型可以维持一个相对合理的复杂度。词序列经由词嵌入层，得到词嵌入向量序列$\{\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_t\}$。

3. RNN

前一个嵌入向量映射层产生的输出，将作为后续 RNN 的输入。这个 RNN，依次以序列的各个时刻的元素$\boldsymbol{e}_t$作为输入，同时以其前一时刻输出产生的隐藏状态变量$\boldsymbol{h}_{t-1}$作为输入，通过 RNN 映射H，得到当前时刻的隐藏状态变量$\boldsymbol{h}_t$，如下：

$$\boldsymbol{h}_t = H(\boldsymbol{e}_t, \boldsymbol{h}_{t-1}) \tag{2}$$

RNN 映射函数H是一个非线性映射，它可以有多种多样的选择。最简单的 RNN 如下：

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_{eh}\boldsymbol{e}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h) \tag{3}$$

其中，$\boldsymbol{W}_{eh} \in \mathbf{R}^{d_h \times d_{\text{emb}}}$，$\boldsymbol{W}_{hh} \in \mathbf{R}^{d_h \times d_h}$，$\boldsymbol{b}_h \in \mathbf{R}^{d_h}$是需要学习的网络权重参数；$\sigma()$是非线性激活函数，如 ReLU。公式（3）介绍的$\boldsymbol{h}_t$是属于相对比较简单的非线性映射函数。还有很多更复杂的非线性映射函数可以作为 RNN 的网络映射H。例如，能够较好刻画词序列远距离依存关系的 LSTM，已经在很多应用中得到广泛应用。相关内容已经在第 4 章中进行了讨论，此处不再赘述。

此外，构建两个 RNN，分别以正序和逆序处理输入词序列，也能更好地对词序列的语义进行刻画。这种网络叫作双向递归神经网络（Bidirectional Recurrent Neural Network，BRNN）。对于 BRNN，隐藏状态变量是来自于前向 RNN 和后向 RNN 所生成的两个隐藏状态变量的拼接，如公式（4）所示。BRNN 保持了关于顺序的对称性，并且更加平衡地使用了前向和后向的上下文信息。

$$\overrightarrow{\boldsymbol{h}_t} = \overrightarrow{H}(\boldsymbol{e}_t, \boldsymbol{h}_{t-1}),\ \overleftarrow{\boldsymbol{h}_t} = \overleftarrow{H}(\boldsymbol{e}_t, \boldsymbol{h}_{t+1}),\quad \boldsymbol{h}_t = \text{concatenate}(\overrightarrow{\boldsymbol{h}_t}, \overleftarrow{\boldsymbol{h}_t}) \tag{4}$$

无论是哪种具体的 RNN 模型，我们都可以将其隐藏状态变量$\boldsymbol{h}_t$看成是第 t 个词对应的表达向量。由 RNN 的迭代更新方式（见公式（2））不难看出，每一个隐藏状态变量都直接或间接地包含了词序列中其他词的信息。因此，隐藏状态变量$\boldsymbol{h}_t$会比$\boldsymbol{e}_t$更好地刻画当前词的上下文语义信息。当一个词有多种语义时，上下文信息对于准确理解这个词在当前语境下的含义就变得非常重要。

4. 基于注意力机制的 pooling

RNN 的输出是一个向量序列$\{\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_t\}$，其中$\boldsymbol{h}_t \in \mathbf{R}^{d_h}$。下一步，我们利用基于注意力机制的 pooling 操作，将这个向量序列进一步压缩为一个单一的向量，以便于构造后续的代价函数。本节中，我们把将向量序列压缩为一个单一向量的过程，叫作 pooling（池化）。

最简单直接的压缩方法，是将词序列的最后一个状态变量$\boldsymbol{h}_t$作为整个序列的表达向量（对于 BRNN，这对应于$\text{concatenate}(\overrightarrow{\boldsymbol{h}_t}, \overleftarrow{\boldsymbol{h}_1})$）。我们称这种方式为 last pooling。请注意，根据公式（2），$\boldsymbol{h}_t$是整个序列$\boldsymbol{X}$的一个函数。因此，理论上讲，

$\boldsymbol{h}_t$应该可以刻画整个序列的信息。但是，这种做法往往会引起长距离依赖性问题（long term dependency problem）。当然，LSTM 特殊的结构，使它对长距离依赖性问题相对不敏感。

mean pooling 是另外一种常用的策略，其最终的表达向量是所有中间状态变量的均值，即$\boldsymbol{h} = \frac{1}{T}\sum_{t=1}^{T}\boldsymbol{h}_t$。显而易见，对于 mean pooling 来说，长距离依赖性问题不再是主要问题，因为所有的中间状态变量均被均等地考虑在内。但是，mean pooling 也有一个缺点，就是它不能够区分哪些词对整体词序列的语义有重要影响，而哪些词对整体语义无关紧要。

max pooling 和 mean pooling 比较类似，其最终的表达向量是所有中间状态变量的最大值，即$\boldsymbol{h} = \max(\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_t)$。这种取最大值的操作提供了另外一层非线性变换。一般来说，max pooling 比 mean pooling 的效果更好一些，应用也更广泛一些。即便如此，max pooling 与 mean pooling 有类似的问题，即无法区分各个词对整体语义的重要程度。

针对上述方法的局限性，本算法采用了基于注意力机制的 pooling 操作。它的形式，是将序列产生的所有状态变量以加权求和的方式，得到最终的表达向量。权重越大，表明其对应的词对整体语义越重要。这些权重，是通过一个专门的网络，根据词具体的上下文计算得到的，而非预先指定的静态权重，因此能更好地刻画词在不同语境下对整体语义的重要程度。注意力池化（attention pooling）的数学公式如公式（5）所示。其中，我们称$s(\boldsymbol{h}_t;\theta)$为注意力网络（attention network），它将状态向量$\boldsymbol{h}_t$映射为一个标量值，并用$\exp(s(\boldsymbol{h}_t;\theta))$来刻画第$t$个词对于整个词序列的重要程度。为了使整个词序列各个词对应的重要程度的分数总和为 1，我们对它进行归一化，从而得到$\boldsymbol{\alpha}_t$，称之为注意力（attention）。

$$\boldsymbol{h} = \sum_{t=1}^{T}\alpha_t\boldsymbol{h}_t\ ,\ \ \boldsymbol{\alpha}_t = \frac{\exp(s(\boldsymbol{h}_t;\theta))}{\sum_{t=1}^{T}\exp(s(\boldsymbol{h}_t;\theta))} \tag{5}$$

为了让 attention network 计算得到的 attention 值能够准确地反映各个词对

当前词序列的真实重要程度，有两个关键前提需要满足。第一，它需要一个能准确反映词及其上下文语义的表达向量，这一点是由前述 RNN 来实现的。第二，映射函数$s(\boldsymbol{h}_t;\theta)$要有足够的模型容量（model capacity）去区分重要的词和不重要的词。例如，考虑到状态向量往往只有几百维，那么简单的线性模型不具备足够的容量来刻画词在复杂的语义空间中的重要程度。因此，本算法是采用了一个专门的神经网络作为映射函数$s(\boldsymbol{h}_t;\theta)$的。这个注意力网络的权重，和如上所述的 RNN 的网络权重，在训练过程中一起进行学习和更新。

综上所述，基于注意力机制的 RNN 模型结构如图 6.2 所示，词序列经词嵌入层被映射为嵌入向量序列，再经由 RNN 或 BRNN 映射为状态向量序列，最后通过注意力网络映射为最终表达向量。

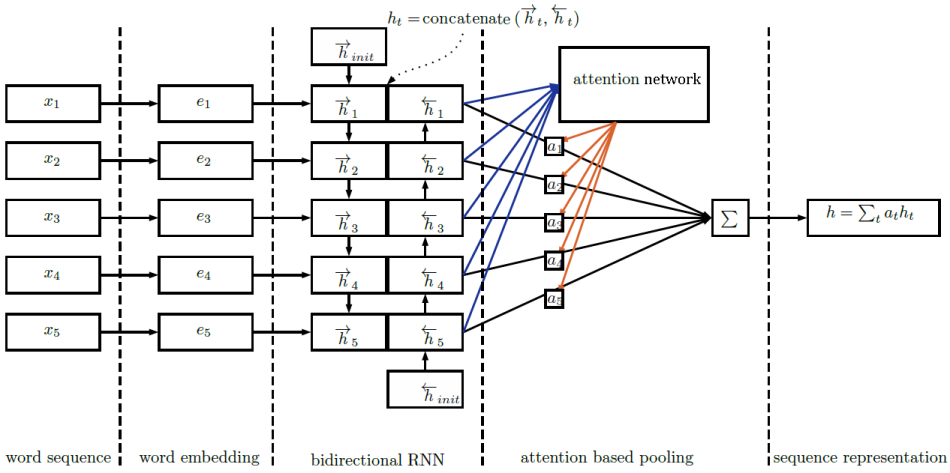


图 6.2　基于注意力机制的 RNN 模型结构

6.2.2　代价函数

如图 6.2 所示，将词序列经过模型映射为最终的表达向量之后，我们需要基于此最终表达向量定义一个合适的代价函数。代价函数是训练学习过程中的目标函数。具体来说，训练过程就是通过调整网络模型中的所有可训练参数（例如，

网络连接中边的权重），实现最小化代价函数的过程。因此，每一个机器学习的训练过程，都需要预先定义好代价函数。

在本章所讨论的实际应用中，我们的目标是，基于用户查询找出相关的广告。什么样的代价函数才能合适地刻画我们这一实际目标呢？又该利用什么样的数据来计算这一代价函数呢？下面，我们就以搜索广告这一具体例子来介绍如何设计代价函数（包括代价函数的数学表达，以及如何利用具体数据计算此代价函数）。

对于商用的搜索引擎系统来说，它往往可以收集到大量的用户行为日志，包括用户输入了哪些查询、点击了哪些广告，等等。通过日志数据，我们可以知道，用户输入查询后，系统展示了哪些广告，并且用户点击了哪些展示的广告。直观地讲，用户点击某个广告，往往说明这个广告与用户输入的查询是相关的；对于一个查询和广告对，用户点击的次数越多，往往说明它们越相关。因此，对任一给定的查询，那些被点击过的广告，都可以被当作训练样本的正例（即相关的广告）来对待。

这里需要注意的是，即使用户并没有点击某个展示的广告，也并不能说明那一对查询和广告完全不相关。有研究表明，即使是和用户查询相关的广告，用户的点击率也仅在 5%~10%以内。因此，对于一个用户查询，我们不能简单地把系统展示了却没有被用户点击的广告作为负例（即无关的广告）来对待。事实上，我们往往是从系统储存的所有广告中随机抽选 N（例如 N=4）个广告作为训练样本集合中的负例。由于系统存储的所有广告的数量往往非常多，随机抽取的广告有非常大的概率是不相关广告。因此，以这种方式构建负例是合理的。此外，为了谨慎起见，在挑选负例时，我们也可以在随机抽取得到的广告中，进一步检查并去除和此查询对应的被点击过的广告。

我们以（q, d^+）表示被点击过的（查询，广告）对，以$\{d_1^-, d_2^-, \ldots, d_n^-\}$表示随机抽取出来的和当前查询$q$不相关的广告。我们以$h_q(x)$和 $h_d(y)$分别表示将查询x

和广告y映射为其最终表达向量的函数。我们以公式（6）来定义代价函数。

$$J(\theta) = -\sum_{(q,d^+)} \log \frac{\exp(h_q(q)^{\mathrm{T}} \times h_d(d^+))}{\exp\left(h_q(q)^{\mathrm{T}} \times h_d(d^+)\right) + \sum_{i=1}^{n} \exp\left(h_q(q)^{\mathrm{T}} \times h_d(d_i^-)\right)} \quad (6)$$

其中，参数$\theta = \{W_{q,\mathrm{emb}}, \theta_{q,\mathrm{rnn}}, \theta_{q,\mathrm{attention}}, W_{d,\mathrm{emb}}, \theta_{d,\mathrm{rnn}}, \theta_{d,\mathrm{attention}}\}$，表示了前述各个子模型所包含的所有参数。$W_{q,\mathrm{emb}}$和$W_{d,\mathrm{emb}}$分别表示用于处理用户查询的词嵌入层和处理广告的词嵌入层的模型参数。$\theta_{q,\mathrm{rnn}}$和$\theta_{d,\mathrm{rnn}}$分别表示用于处理用户查询和广告的 RNN 的参数。而$\theta_{q,\mathrm{attention}}$和$\theta_{d,\mathrm{attention}}$分别表示用于处理用户查询和广告的注意力网络的参数。

实际的训练学习过程如下：根据每一对点击的查询广告对（q, d^+），随机抽取n个没有被查询q点击过的广告（$d_1^-, d_2^-, \ldots, d_n^-$），进而可以根据已有的参数$\theta$依次分别计算出查询$q$和广告$d^+, d_1^-, d_2^-, \ldots, d_n^-$各自对应的文本表达向量、RNN 的状态向量，以及经过注意力网络后得到的最终表达向量。基于它们各自的最终表达向量，就可以代入公式（6）计算出这一对点击过的查询广告对所对应的代价函数值。我们对所有点击过的（查询，广告）对计算其各自的代价函数值，并进行加和，就可以得到在整个样本集上的代价函数值。

通过优化方法，可以不断调整模型中各个参数的值，从而使整体的代价函数值最小。而对应这个整体代价函数值最小的那组参数，就是模型的最终参数。一般来说，SGD 是现在使用最广泛的优化方法。

【知识点讲解】我们需要说明的是，公式（6）这种代价函数的形式被广泛地应用在很多文本处理相关的领域。具体说来，它以 softmax 来表达后验概率，并用后验概率的 log-loss 之和作为整体的代价函数。我们可以这样解读公式（6）：对于样本$x = q$，需要从$n+1$个类别（$d_1^-, d_2^-, \ldots, d_n^-, d^+$）中，预测出它所属的真正类别$y = n+1$（即$d^+$）。这是一个多类别分类问题。当我们用 softmax 处理多类别分类问题时，往往是将样本q属于第i类的后验概率表达为公式（7）。

$$P(y = i | x = q) = \frac{\mathrm{e}^{z_i}}{\sum_{k=1}^{n+1} \mathrm{e}^{z_k}} \quad (7)$$

其中，z_i是一个分数，用以刻画样本q和类别i之间的相似程度。之所以把公式（7）的形式称为 softmax，是因为根据它所计算得到的各个类对应的后验概率中，往往只有分数最高的那个类具有很高的后验概率（比较接近 1），而其他各个类所对应的后验概率都非常低（往往比较接近 0），其作用就好比进行求最大值（max）的操作一样，但又不是像求最大值操作那样进行“硬编码”（即具有最大分数的类后验概率为 1，其他都为 0），而是具有一定平滑作用的“软编码”，所以叫作 softmax。对比公式（6）及公式（7），我们不难看出，公式（6）相当于使用$\boldsymbol{z}_i = h_q(q)^{\mathrm{T}} \cdot h_d(d_i)$来计算 softmax。也就是使用查询的表达向量和广告的表达向量的内积（也叫数量积或点积）作为分数 z；内积越大，则说明这两个表达向量越类似，其对应的后验概率就越高。当计算出用户查询q对应于每个广告的后验概率之后，很自然地，我们可以根据最大后验概率准则作为优化的目标函数。在实际中，为了计算方便，我们往往使用后验概率的负对数求和来作为目标函数，也就是如公式（6）所示的代价函数。因此，公式（6）所示的代价函数，是以最大后验概率为理论基础，用 softmax 对后验概率进行建模，从而推导出来的。在其他的分类问题中，也依然可以采用类似的方式进行建模，构造合适的代价函数作为学习目标。

6.3 应用 TensorFlow 实现 DeepIntent 模型

本节将介绍如何用 TensorFlow 实现 DeepIntent 模型。需要说明的是，考虑到篇幅限制，我们只会挑出一些重要的部分加以讲解。因此示例代码并非是可以完整编译运行的代码。

6.3.1 定义计算图

第一步是定义从原始样本输入到高维特征向量输出的计算流程。TensorFlow 会根据我们定义的计算流程，自动生成相应的计算图。

1. 输入层的实现

首先，定义整个模型的输入。在 TensorFlow 中，一般使用 tf.placeholder 来定义输入。需要注意的是，当我们在定义计算图时，用 tf.placeholder 来定义的输入变量并不会被赋值，只是相当于一个占位符。它告诉编译器，在实际执行这个计算图时输入数据的维度和数据类型等信息。只有在程序实际运行时才会对 tf.placeholder 定义的输入进行赋值，输入实际的数据。

在以下的所有示例代码中，为简便起见，所有外部传入的可变参数，均以“FLAGS.”为前缀来标识。并且，我们根据变量表示的实际意义为变量命名，便于理解代码。

如下示例定义了 3 个输入，query_tensor、offer_tensors, 以及 click_tensors。它们分别对应着用户查询、广告，以及用户的点击信息。其中，dtype 表示此输入变量的数据类型，shape 表示输入变量的维度，name 表示输入变量的名字。

以 query_tensor 为例，它的 shape=(None, query_rnn_length)，这表示它是一个二维矩阵。

第一维的值为 None，表示矩阵的第一维的长度可变；它是在程序运行时，根据实际的输入数据来决定的。事实上，由于几乎所有的神经网络的训练都是基于 minibatch 的，因此，输入变量的第一维，一般都是对应于 minibatch 中含有的样本个数（我们用参数 FLAGS.batch_size 来指定 minibatch 中含有的样本个数）。因此在定义输入样本数据时，往往把第一维的值设为 None 或者 FLAGS.batch_size。

query_tensor 的第二维是 FLAGS.query_rnn_length，表示用户查询字符串的长度。在本程序中，query_tensor 所对应的二维矩阵，其行数对应 minibatch 中的样本个数；而每一个行向量对应着一个样本（即一个输入词序列），行向量中的每个元素是词对应的 id。

```
    query_tensor = tf.placeholder(dtype=tf.int32, shape=(None,
FLAGS.query_rnn_length), name="query_tensor")
    offer_tensors = tf.placeholder(dtype=tf.int32, shape=(None,
FLAGS.offer_rnn_length), name="offer_tensor")
    click_tensors = tf.placeholder(dtype=tf.float32, shape=(None),
name="query_offer_clicks")
```

2. 词嵌入层的实现

除输入层以外，第一个计算层是词嵌入层（word embedding layer），参见 6.2.1 节。下面，我们看一下如何实现词嵌入层的计算逻辑。TensorFlow 中已经提供了大量封装好的函数，可以方便地实现各种常见的网络结构。tf.contrib.layers.embed_sequence 即可实现将输入的词序列映射为词嵌入向量序列。这一函数调用，创建了一个大小为[vocab_size, embed_dim]的变换矩阵（对应于 6.2.1 节中的$\boldsymbol{W}_{\text{emb}}$），并将词的 id 映射为维数为 embed_dim 的嵌入向量。因此，维数为[FLAGS.batch_size, FLAGS.query_rnn_length]的输入 query_tensor，经过 tf.contrib.layers.embed_sequence 函数映射后，得到维数为[FLAGS.batch_size, FLAGS.query_rnn_length, FLAGS. embedding_size]的输出 word_vectors。

```
    word_vectors = tf.contrib.layers.embed_sequence(query_tensor, vocab_size=
query_word_number, embed_dim=FLAGS.embedding_size, initializer=
tf.random_normal_initializer(mean=0.0, stddev=FLAG.scale_rate))
```

在函数 tf.contrib.layers.embed_sequence 的参数中，还可以通过参数 initializer 指定初始化函数。如前所述，tf.contrib.layers.embed_sequence 创建了变换矩阵$\boldsymbol{W}_{\text{emb}}$。而这个矩阵通过 initializer 参数所指定的函数进行初始化。这里，tf.random_normal_initializer 是 TensorFlow 提供的正态分布初始化函数，即它根据正态分布来随机产生矩阵的元素值，而正态分布的均值被指定为 0，标准方差为 FLAG.scale_rate。

【知识点讲解】在定义神经网络的计算流程时，会涉及很多运算及相应的网络单元（也即模型变量）。如果简单地把这些模型变量置 0 进行初始化，则会导

致整个模型的初始状态对任何输入数据均难以响应，因而无法根据梯度下降对模型进行有效的优化。因此，在实际应用中往往需要用随机数对模型变量进行初始化。在创建网络单元（也即模型变量）时，TensorFlow 提供的 API 大多由初始化函数作为参数，以便用户可以指定特定的初始化方式。当然，在创建各个网络单元时，我们也可以不特别指定其初始化函数。然后在模型训练之前，通过运行初始化函数 tf.global_variables_initializer，将定义的所有模型变量进行统一的初始化。此时，那些特别指定了初始化函数的模型变量，仍将使用其指定的初始化函数进行初始化。那些没有特别指定初始化函数的模型变量，将被 tf.global_variables_initializer 函数初始化。

【知识点讲解】计算图的概念对于 TensorFlow 至关重要。在计算图中，所有的运算都被表示成计算图上的一个节点，而节点上的入边表示输入数据，出边表示输出数据。因此，数据就好像是在计算图上定义的各个计算节点之间依次流动，直至生成最终的输出结果。这也是 TensorFlow 名字的由来。在利用 TensorFlow 构建计算图时，只需要定义好从输入到输出的计算流程即可，不需要关注如何求梯度、如何更新网络参数的权重等问题。当定义好从输入到输出的计算流程之后，TensorFlow 会将相应的计算节点加入计算图。从输入到输出的计算流程，对应着计算图中前向网络（forward network）部分。根据定义好的前向网络，TensorFlow 会自动在计算图中添加一系列的辅助计算节点（如求导数运算等），用以计算反向传播（back propagation）需要的所有相关计算。注意，反向传播所需的计算操作都是 TensorFlow 自动生成的，不需要用户自己定义。这种设计使得用户可以专注于定义模型本身，而不需要关注利用链式法则求导及更新参数等具体实现细节。

3. RNN 的实现

下面，我们介绍 RNN 的实现。请注意，RNN 是由一个或多个具有相同结构的基本单元拼接而成的。不同类型的基本单元，构成了不同类型的 RNN。

TensorFlow 提供了多种常见的 RNN 基本单元，例如 GRU 网络单元、LSTM 网络单元等。

在实现 RNN 时，需要先定义 RNN 的基本单元。为此，我们定义了函数 get_cell_fn。可以通过指定参数 cell_type 的值，来定义不同类型的基本单元。以 GRU 为例，通过 TensorFlow 的一个函数调用 tf.contrib.rnn.GRUCell，即可创建一个 GRU 基本单元。其中，参数 num_units 表示 RNN 基本单元的输入数据的维数。在本例中，由于 RNN 是以词嵌入层的输出作为输入的，因此，num_units 的值应为 FLAGS. embedding_size。

```
def get_cell_fn(cell_type='gru', num_units):
        cell_fn = tf.contrib.rnn.GRUCell(num_units)
        if cell_type == 'rnn':
            cell_fn = tf.contrib.rnn.BasicRNNCell(num_units)
        elif cell_type == 'gru':
            cell_fn = tf.contrib.rnn.GRUCell(num_units)
        elif cell_type == 'lstm':
            cell_fn = tf.contrib.rnn.BasicLSTMCell(num_units, state_is_tuple=True)
        return cell_fn
```

在定义好 RNN 中的基本单元之后，可以创建一个基本单元数组 cell_list，并调用函数 tf.contrib.rnn.MultiRNNCell 将 cell_list 所包含的多个基本单元组合成最终的 RNN。

```
cell_list = [get_cell_fn(cell_type=FLAGS.cell_type, FLAGS.embedding_size) for _
in range(FLAGS.num_rnn_layers)]
rnn_network = tf.contrib.rnn.MultiRNNCell(cell_list)
```

在定义好 RNN 之后，还需要定义基于 RNN 的运算操作，也就是将前一层（词嵌入层）的输出作为 RNN 的输入，通过计算得到 RNN 的输出。实现这一运算操作也非常简单。给定一个 minibatch 作为输入（inputs=word_vectors），并指定各个输入样本需要处理的长度 document_lengths（它是一个长度为 FLAGS.batch_size 的数组，每个元素表示 minibatch 中对应样本的字符串长度），通过调用函数 tf.nn.dynamic_rnn 就可以计算一个 RNN（cell=rnn_network）的

最终输出状态（rnn_states），以及其中间状态结果（rnn_outputs）。其中，rnn_states 的大小为[FLAGS.batch_size，cell_state_size]，rnn_outputs 的大小为[FLAGS.batch_size.batch_size，max_length，cell_state_size]。其示例代码如下。

```
    rnn_outputs, rnn_states = tf.nn.dynamic_rnn(cell=rnn_network,
inputs=word_vectors, sequence_length=document_lengths, dtype=tf.float32)
```

在 RNN 用于自然语言理解和处理的相关应用中，双向 RNN 往往可以达到比单向 RNN 更好的结果。这是因为双向 RNN 可以充分地编码前向上下文以及后向上下文，而单向 RNN 往往只能编码单方向的上下文。下面，我们介绍一下双向 RNN 的实现。事实上，双向 RNN 与单向 RNN 的实现大体思路比较类似，只有具体的函数调用有所区别。整体流程是，先定义 RNN 的基本单元，然后分别创建前向 RNN 及后向 RNN，之后调用特定的函数，在计算给定输入的情况下，网络的输出及中间状态。

可以看到，我们用 tf.contrib.rnn.MultiRNNCell 分别将多个 RNN 基本单元组合为前向 RNN 及后向 RNN，然后调用函数 tf.nn.bidirectional_ dynamic_rnn 来计算双向 RNN 在给定输入为 inputs=word_vectors 的情况下，此双向 RNN 的最终输出状态（rnn_states）及其中间状态结果（rnn_outputs）。最后，通过 tf.concat 函数将前向网络的输出以及后向网络输出的中间状态结果拼接起来。

```
    forward_cell_list = [get_cell_fn(cell_type=FLAGS.cell_type, FLAGS.embedding_size)
for _ in range(FLAGS.num_rnn_layers)]
    backward_cell_list = [get_cell_fn(cell_type=FLAGS.cell_type,
FLAGS.embedding_size) for _ in range(FLAGS.num_rnn_layers)]
    forward_rnn_network = tf.contrib.rnn.MultiRNNCell(forward_cell_list)
    backward_rnn_network = tf.contrib.rnn.MultiRNNCell(backward_cell_list)
    rnn_outputs, rnn_states = tf.nn.bidirectional_dynamic_rnn(cell_fw=
forward_rnn_network, cell_bw= backward_rnn_network, inputs=word_vectors,
sequence_length=document_lengths)
    rnn_outputs = tf.concat(rnn_outputs, 2)
```

4. 基于注意力机制的池化层的实现

根据 6.2.1-4 节所述，我们将构建一个注意力网络，以 RNN 输出的中间状态结果 rnn_outputs 为输入，通过计算，得到词序列中各个词的注意力分数。这个注意力分数是注意力网络的输出，体现各个词的权重。在计算得到各个词的权重之后，进一步将所有中间状态结果（即 rnn_outputs）按照对应的权重进行加权平均，得到最后的映射向量。

下面，我们先讲解注意力网络的实现，即 6.2.1-4 节中的映射$s(\boldsymbol{h}_t;\theta)$。为实现注意力网络，我们需要定义如下几个函数。

```
   def hidden_layer(filter_shape, x, activation=None):
          hid_w = tf.Variable(tf.random_normal(shape=[filter_shape[1],
filter_shape[0]], mean=0.0, stddev=1e-2, dtype=tf.float32, seed=123, name=None),
name="hid_w")
          hid_b = tf.Variable(tf.zeros([filter_shape[0]]), name="hid_b")
          hid_lin = tf.matmul(x, hid_w) + hid_b
          if activation:
              return activation(hid_lin)
          else:
              return hid_lin
```

首先定义函数 hidden_layer，可以看出，这是一个一层的全连接神经网络。filter_shape 是一个包含两个元素的向量，第一个元素的值为输入特征的维度，第二个元素的值为输出层神经元的个数。hid_w 是全连接网络中，从输入特征到输出层神经元的网络连接的权重。hid_b 是输出层神经元的偏置向量。

下面，我们定义函数 attention_layer。

```
   def attention_layer (rnn_outputs, filter_shape, mask, maximum_document_length):
          if FLAGS.bidirectionalRNN:
              embedding_size = FLAGS.embedding_size * 2
          else:
              embedding_size = FLAGS.embedding_size
          rnn_outputs_reshapeTo2D = tf.reshape(rnn_outputs, [-1, embedding_size])
          filter_shape2 = [1, filter_shape[0]]
          hid =hidden_layer(filter_shape, rnn_outputs_reshapeTo2D,
activation=tf.nn.relu)
          y_raw = hidden_layer(filter_shape2, hid, activation=None)
```

```
    y_reshape = tf.reshape(y_raw, [-1, maximum_document_length, 1])
    y_reduce_max = tf.reduce_max(y_reshape, axis=1, keep_dims=True)
    y1 = tf.subtract(y_reshape, y_reduce_max)
    mask = tf.reshape(mask, [-1, maximum_document_length, 1])
    y2 = tf.multiply(tf.exp(y1), mask)
    attention_p = tf.divide(y2, tf.reduce_sum(y2, axis=1, keep_dims=True))
    attention_outputs = tf.multiply(rnn_outputs, attention_p)
    attention_pooling_outputs = tf.reduce_sum(attention_outputs, axis=1)
    return attention_outputs, attention_pooling_outputs, attention_p
```

这个函数以 RNN 输出的所有中间状态向量为输入，实现了 6.2.1-4 节所述的注意力网络的计算逻辑，并得到最终的输出结果。函数的整体流程比较简单明了，即通过两次调用 hidden_layer，实现了一个两层全连接网络，将输入的 RNN 层的中间状态向量映射为 y_raw；它的每个元素，对应着公式（5）中的$s(\boldsymbol{h}_t;\theta)$。而后，$\boldsymbol{y}1$ 是将结果减去其最大值，使得所有的$s(\boldsymbol{h}_t;\theta)$值均是小于或等于 0 的值，这样可以避免在随后$\boldsymbol{y}2$ 计算$\exp(s(\boldsymbol{h}_t;\theta))$时发生数值计算溢出。然后，根据公式（5），计算得到 attention_p，它的每个元素是对应词的 attention score。以各个词的注意力分数（attention score）作为权重，对 RNN 各个词对应的中间状态向量进行加权平均，即得到了最终的表达向量 attention_pooling_outputs。除上述的操作外，我们调用了几次 tf.reshape。这一函数调用，只是改变数据的形状，以便于进行必要的矩阵乘法或函数调用操作。

至此，我们实现了 DeepIntent 模型的构建，即定义了从输入词序列一步一步计算得到词序列的最终表达向量。

6.3.2 定义代价函数及优化算法

在 6.3.1 节中，我们定义了如何将输入词序列映射为最终表达向量。有了词序列的最终表达向量，还需要定义如何根据最终表达向量来计算代价函数，并指定相应的优化算法，这样 TensorFlow 才能够自动根据优化算法对模型中所有涉及的可训练参数进行调整，实现对代价函数的优化。

事实上，根据最终表达向量，通过进一步计算，得到代价函数的值，这个“代

价”才是整个计算图最终的输出结果。因此，从输入词序列映射到最终表达向量的计算流程，以及从最终表达向量映射到代价值的计算流程，都属于计算图的一部分。TensorFlow 会把它们一起编译为计算流程图。但是，从机器学习的概念上讲，前一部分对应于传统的特征学习和提取，即把原始输入样本转换为高维空间中的一个特征向量；后一部分的优化目标往往是和特征提取相对独立的部分。因此，我们在这里分开介绍。

下面，我们介绍如何实现代价函数。根据 6.1 节所述流程，可以得到任何一个输入词序列对应的最终表达向量。我们将用户查询对应的最终表达向量记为 query_encoding，广告对应的最终表达向量记为 offer_encodings。对于一对<用户查询，广告>，它被用户点击过多少次这一信息存储在 click_tensors 里。DeepIntent 模型中所定义的代价函数的具体形式如 6.2.2 节公式（6）所示。为计算它，我们先定义 train_similarity_score 函数，用来计算两个向量的内积相似度（即公式（6）中的$h_q(q)^{\mathrm{T}} \cdot h_d(d_i)$）。需要说明的是，由于在训练时，每一个用户查询会和一个点击过的广告组成一个正例，同时它会和 FLAGS.negative_offer_number 个随机选取的未点击过的广告组成负例，因此在计算相似度时，我们需要先对 query_encodings 及 offer_encodings 进行重塑（reshape），以使得其维数正确地对应着前述逻辑。具体代码如下所示。

```
   def train_similarity_score(query_encodings, offer_encodings):
      with tf.name_scope('Score_Variant'):
         expanded_query_encodings = tf.tile(query_encodings, [1,
FLAGS.negative_offer_number + 1])
         if FLAGS.bidirectionalRNN:
            expanded_query_encodings = tf.reshape(expanded_query_encodings,
[-1,2*FLAGS.embedding_size])
         else:
            expanded_query_encodings = tf.reshape(expanded_query_encodings,
[-1,FLAGS.embedding_size])
         if FLAGS.score_variant == "cossim":
            score_variant = calculate_cosine_sim(expanded_query_encodings,
offer_encodings, name_scope="Training_Cosine_Similarity",use_normalization=True)
```

```
            score_variant = tf.reshape(score_variant, [-1,
FLAGS.negative_offer_number + 1])
        elif FLAGS.score_variant == "dotprod":
            prod = calculate_cosine_sim(expanded_query_encodings, offer_encodings,
name_scope="Training_Dot_Product",use_normalization=False)
            prod = tf.reshape(prod, [-1, FLAGS.negative_offer_number + 1])
            score_variant = tf.subtract(prod, tf.reduce_max(prod, axis=1,
keep_dims=True))
        return score_variant
```

代价函数的定义如下所示。

```
    def get_loss(query_encoding, offer_encodings, click_tensors):
        cosine_similarity =train_similarity_score(query_encoding, offer_encodings)
        target = tf.zeros(FLAGS.batch_size, tf.int32)
        target = tf.one_hot(target, (FLAGS.negative_offer_number + 1), 1, 0)
        if FLAGS.click_weighting == "log2" or FLAGS.click_weighting == "log2_weights":
            batch_weights = 1.0
            if FLAGS.use_click_count:
                if FLAGS.click_weighting == "log2":
                    batch_weights = click_tensors
                elif FLAGS.click_weighting == "log2_weights":
                    batch_weights = tf.log1p(click_tensors)
            loss = tf.losses.softmax_cross_entropy(onehot_labels=target,
logits=cosine_similarity, weights=batch_weights)
        return loss
```

其中，target 定义了目标，相当于样本标注，即点击过的<用户查询，广告>对被标记为 1，未点击过的<用户查询，广告>对被标记为 0。click_tensors 里存储着点击次数的信息。如果我们不考虑使用这一信息（如公式（6）所示），那么直接将 batch_weights 设定为 1.0，然后调用 tf.losses.softmax_cross_entropy 来计算代价值。如果我们希望使用用户点击次数这一信息，即获得点击次数越多的<用户查询，广告>对，其权重越高，那么就可以通过设置 batch_weights 来达到这一目的。

定义好代价函数之后，需要告知 TensorFlow 优化的目标就是这个代价函数。此外，还需要告知 TensorFlow 具体选用的优化算法是什么。这些逻辑可以通过如下所示函数实现。

```
    loss = get_loss(query_encoding, offer_encodings, click_tensors)
    global_step = tf.Variable(0, name="global_step")
    optimizer = get_optimizer(FLAGS.optimizer_name, FLAGS.learning_rate)
    train_op = optimizer.minimize(loss, global_step=global_step)
```

其中，optimizer 返回了由程序参数指定的最优化算法，其具体实现如函数 get_optimizer 所示。而 train_op 具体定义了一种操作，这一具体操作将最优化算法应用于最小化我们的目标函数，即代价函数值 loss。请注意，train_op = optimizer.minimize(loss, global_step=global_step) 这一语句只是定义了 train_op 这一操作，表明 train_op 这一操作所对应的逻辑是“利用 oprimizer 所代表的最优化算法，根据最小化 loss 值这一目标，去更新所有相关的模型参数”。再次提醒读者朋友注意，计算图的定义，只是定义从输入到输出的计算流程，以及相应最优化操作的逻辑。此刻，系统还未进行任何实际的操作和计算。

```
    def get_optimizer(optimizer_name='gradientdescent', learning_rate=0.001):
        optimizer_fn =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
        if optimizer_name == 'gradientdescent':
            optimizer_fn =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
        elif optimizer_name == 'adadelta':
            optimizer_fn = tf.train.AdadeltaOptimizer(learning_rate=learning_rate,
epsilon=1e-6)
        elif optimizer_name == 'adam':
            optimizer_fn = tf.train.AdamOptimizer(learning_rate=learning_rate)
        elif optimizer_name == 'momentum':
            optimizer_fn = tf.train.MomentumOptimizer(learning_rate=learning_rate,
momentum=0.9)
        elif optimizer_name == 'adagrad':
            optimizer_fn = tf.train.AdagradOptimizer(learning_rate=learning_rate)
        elif optimizer_name == "rmsprop":
            optimizer_fn = tf.train.RMSPropOptimizer(learning_rate=learning_rate)
        elif optimizer_name == "ftrl":
            optimizer_fn = tf.train.FtrlOptimizer(learning_rate=learning_rate,
l1_regularization_strength=0, l2_regularization_strength=0)
        return optimizer_fn
```

6.3.3 执行计算图进行训练

在 6.3.2 节中，我们实现了 DeepIntent 模型的计算图，定义了如何从输入通过词嵌入层、RNN、基于注意力机制的 pooling 层一步一步映射为最终表达向量，也定义了如何从最终表达向量进一步计算得到最终的代价值，并定义了如何用优化算法对模型参数进行更新来最小化代价值的操作。至此，模型的逻辑已经定义完备。下一步，就是具体的执行优化操作，一步一步地更新模型参数，直至收敛，完成训练。

要想真正运行计算图中所定义的操作，首先需要创建一个 Session 对象。然后通过 session.run 函数来运行相应的操作和运算。具体来说，我们需要循环执行 6.3.2 节中定义的 train_op 操作，用来更新模型参数，直至收敛完成模型的训练。这一逻辑通过如下代码完成。

```
   saver = tf.train.Saver()
   local_step = 0
   with tf.Session() as sess:
       for batch_idx in range(batch_num):
           local_step += 1
           query_tensor1, offer_tensors1, clicks_tensors1, query_lengths1,
offer_lengths1, query_mask1, offer_mask1 = pull_batch(batch_idx,
query_sentences,offer_sentences, train_clicks_q, train_clicks_o, train_clicks_q_o,
offer_size, query_origin_sentence_lengths, offer_origin_sentence_lengths,
query_mask, offer_mask)
           _, loss_value = sess.run([train_op, loss],feed_dict={query_tensor:
query_tensor1,offer_tensors: offer_tensors1, click_tensors: clicks_tensors1,
query_lengths: query_lengths1, offer_lengths: offer_lengths1, query_mask_tensor:
query_mask1, offer_mask_tensor: offer_mask1})
           if local_step % 20 == 1:
               local_total_loss += loss_value
               saver.save(sess, os.path.join(FLAGS.model_dir, "model.ckpt"),
global_step)
```

在上述代码中，首先创建了 tf.Session()对象 sess，并对每一个 batch，通过调用 pull_batch 准备其对应的输入数据。然后，将输入数据通过 feed_dict 赋值给计算图中对应的输入数据节点，并调用 sess.run 函数来执行我们想要的计算图的节点。在这个例子中，我们执行计算图中的两个节点，即 train_op 和 loss。请

注意，计算图中定义的节点，有的是属于数据节点，如 loss，它是对应了一个 tensor；有的是属于操作节点，如 train_op，它实际是对应了一种操作。无论是数据节点，还是操作节点，当我们把它们放到 session.run()中进行执行时，TensorFlow 就会根据计算图，自动推演出所需的计算流程及其依赖的输入数据，并依此进行执行。也只有在这个时候，计算图所定义的运算及操作，才真正被运行。

在上面这个示例中，我们还定义了 saver 对象，它是用来保持模型的中间训练结果的。在本例中，每隔 20 步，就保存一下模型的当前状态，这样可以避免因训练出错或系统重启等原因而使得模型状态彻底丢失，使得之前所有训练计算被白白浪费。

6.4 本章小结

本章介绍了基于注意力机制的 RNN 模型的深度学习网络在信息检索领域的应用。具体介绍了问题的应用背景、具体的 DeepIntent 算法，以及如何用 TensorFlow 快速实现这一算法。

下面回顾一下关于算法方面的要点。

- 分析实际问题，进行抽象和建模。如本章中，我们将信息检索问题抽象为一个相关性判决问题，即判定任意给定的一个<用户查询，广告>对是否相关。此外，还利用用户点击行为，作为相关性的标注信息加以利用。这样，问题就转化为一个有监督的二分类问题。
- 在用深度学习解决实际问题时，往往先要根据实际问题的特点，选用合适的网络结构，将原始输入样本通过神经网络映射为高维空间中的特征向量。如在本章中，考虑到原始样本（用户查询及广告）均是字符串，因此采用 RNN 这一基本结构进行处理，因为它能够有效地处理序列信息。又

考虑到文本序列自身的语义特点，进一步引入了注意力机制，用于刻画词序列中那些关键部分，并降低那些非关键部分的影响。

- 基于特征向量设计合适的代价函数。这一代价函数应该能合理地衡量预测结果及真实标注之间的差异。本章中所使用的最大熵准则是在文本处理领域非常常用的准则。

参考文献

[1] ZHAI S, CHANG K H, ZHANG R, et al. DeepIntent: Learning Attentions for Online Advertising with Recurrent Neural Networks. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining [C].s, 2016:1295-1304.

第 7 章

图像识别及在广告搜索方面的应用

读者可以想象一个场景：一家广告服务商为视觉产品识别提供了一个广告搜索的 API。第三方网站可以利用 API，从其所拥有的图片中获得产品推荐，或者用户从拍摄的照片中直接获得产品推荐。通过向 API 发送图片，第三方网站可以收到多个视觉上相似的产品广告建议，从而向用户推荐。用户通过点击广告来实现他们的购物意图。这样一来，第三方网站不仅可以提供以图搜图的用户体验，同时还能与广告 API 的提供者分享广告点击营收。在本章中，我们将介绍此 API 背后的技术和一个实现案例：利用 GoogLeNet DNN 将图像编码为浮点矢量，并从数十亿的产品广告中应用近似最近邻（ANN）搜索。此外，我们还会介绍如何减少编码矢量的大小进而加速 ANN 搜索的工作，以及用于过滤不太相关的广告的精确层。最后，我们将实现一个可以通过手机调用 API 的 iOS 应用程序。

7.1 视觉搜索

视觉搜索（以图搜图）是 IT 企业巨资投入的领域，如 Pinterest[1]、eBay[2]、Alibaba[3]和 Microsoft Bing[4]等。视觉搜索源自基于内容的图像检索（CIBR）的研究领域，主要用于输入图像后搜索到在视觉上呈相似项目的排序列表。这对于电

1 Jing, Y., Liu, D., Kislyuk, D., Zhai, A., Xu, J., Donahue, J., & Tavel, S. Visual Search at Pinterest. Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining . New York, NY, USA1,2015: 1889-1898.

2 Yang, F., Kale, A., Bubnov, Y., Stein, L., Wang, Q., Kiapour, H., & Piramuthu, R. Visual Search at eBay. Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining New York, NY, USA2,2017: 2101-2110.

3 Zhang, Y., Pan, P., Zheng, Y., Zhao, K., Zhang, Y., Ren, X., & Jin, R. Visual Search at Alibaba. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. New York, NY, USA0,2018: 993-1001.

4 Hu, H., Wang, Y., Yang, L., Komlev, P., Huang, L., Chen, X. (., . . . Sacheti, A. Web-Scale Responsive Visual Search at Bing. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining . New York, NY, USA3,2018: 359-367.

子商务领域来说尤其有价值，用户可以在系统中通过图片搜索产品，系统也可以向用户推荐基于图像内容的产品。

将视觉搜索搭建于产品广告之上，可以带来更独特的价值主张。若能在第三方网站利用视觉搜索推荐数十亿个可获利的产品广告，这不仅可以为第三方网站带来收入，还能提供额外更具吸引力的用户体验。如图 7.1 所示，如果用其图片调用视觉搜索广告 API，则可以在底部显示多个视觉上类似的广告。

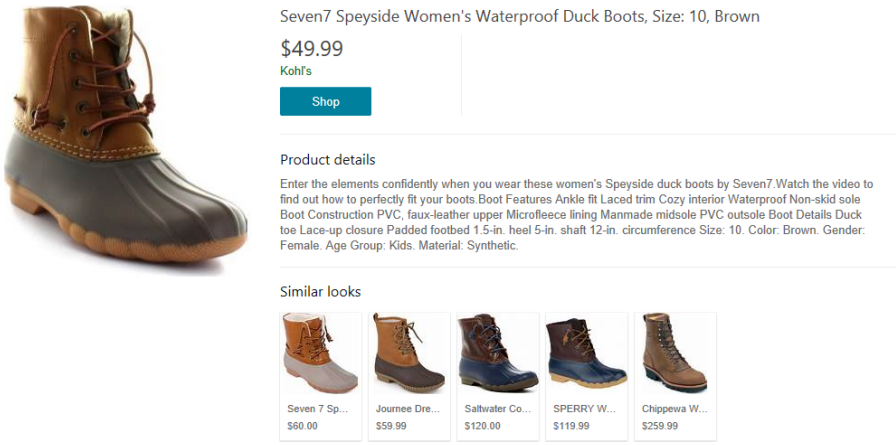


图 7.1　视觉搜索广告

本章主要介绍视觉搜索 API 背后的技术堆栈，其中的组件旨在解决为许多第三方发布者提供服务的挑战。

- 准确率：我们利用 GoogLeNet（亦称为 Inception V1 ）来理解搜索图像。GoogLeNet 模型是 ImageNet Challenge（ILSVRC）[1]的最新突破，对图像分类的准确率已经达到人类的水平。通过搜索图像的编码矢量进而用于产品广告中的 ANN 搜索。每个产品广告图像也使用相同的 GoogLeNet

1 Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S,, . . . Li, F-F. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 115, 2015：211-252.

模型编码。在查询点击信息和文本元数据的帮助下，在 GoogLeNet 模型中对 1024 维编码向量进行降维，得到的 64 维编码矢量可用于快速选择广告。此外，我们还引入了精确图层，以进一步控制推荐广告的质量。该图层利用产品文本元数据和图像编码置信度得分来使其仅保留相关广告。

- 规模：为了在数十亿产品广告中快速搜索，我们为基于矢量的最近邻搜索构建分布式分区索引，聚合每个分区的结果以通过 API 返回。我们还利用分散系统来分配和负载平衡图像编码请求，这是工作流中最昂贵的部分。

7.2 方法和系统

7.2.1 图像 DNN 编码器

我们尝试了几个 DNN 架构，包括 ZFSPPNet[1]、GoogLeNet[2]等模型。首先将输入图像表示为编码向量，并同时将一些采样的广告图像表示为嵌入向量，然后观察向量上最短距离的广告图像是否与输入向量相似。我们观察到 GoogLeNet 的表现优于 ZFSPPNet，尤其是在被裁剪下来的小图像上识别产品时。最后，我们使用 Bing 搜索团队在 Imagenet22K 数据集上训练的 GoogLeNet 模型。模型的细节可参照 Szegedy 所写的文章得到，而训练的过程是在不停地更新模型参数以缩小图像分类的错误。输入图像在 DNN 的最后隐藏层被编码为 1024 维的浮点向量，图像中的信息如产品类型、产品颜色和纹理信息等都已嵌入其中。但是，如此的向量大小对于数十亿产品广告中的 ANN 搜索来说还是太大。因此，我们使用以下的降维技术将 1024 维减少到 64 维以进行进一步的 ANN 搜索。

1 Zeiler, M. D., & Fergus, R. Visualizing and Understanding Convolutional Networks. CoRR,2013.

2 Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Rabinovich, A. Going Deeper with Convolutions. Computer Vision and Pattern Recognition (CVPR), 2015.

7.2.2 利用 Rich-CDSSM 降低维度

本节我们主要利用类似 CDSSM 的模型来降低维度，在此称为 Rich-CDSSM。在产品数据库中，每个产品广告除了图像本身，还包括标题、颜色、适用性别、年龄、卖家名称，以及广告商提供的许多其他字段。不仅可以利用文本信息，而且还可以利用查询和广告点击数据进行降维。用户查询包含许多关键信息，例如产品名称等。当用户基于他输入的查询点击广告时，查询和文本数据都与图像有高度相关。根据此假设，Rich-CDSSM 模型的每个训练样本都包含一个查询和产品的相关信息<查询，产品图像，产品标题，卖方名称>。如图 7.2 所示，模型训练由一方查询引导，另一方提供产品详细信息。

我们首先简单介绍 Rich-CDSSM 如何将文本编码成矢量。简而言之，Rich-CDSSM 模型使用三字母卷积网络将查询变换为 64 维表示向量，如图 7.2 的左半部分所示。查询的每个单词先被切成多个三字母并转换成三字母出现频率向量。举例来说，单词 product 被切成多个三字母——pr，pro，rod，odu，duc，uct，ct。三字母的词汇大小约为 5 万个，因此一个单词可被所有三字母出现频率的 5 万维的矢量表示。接下来，Rich-CDSSM 先使用长度为 3 的卷积网络将每三个连续单词的 5 万矢量加总成 288 维向量序列，再用最大池化（max pooling）将多个 288 维向量总结为一个 288 维矢量以代表整个查询，最后使用一个完全连接投影层将 288 维转至 64 维。运作细节可参照 Huang 所发表的文章。[1]相对于单词嵌入层，三字母向量表示方法可以避免出现未见词汇（out of vocabulary）的问题。

如图 7.2 所示，查询的另一边，每个产品细节（标题、图像和卖方名称）都嵌入 64 维向量，然后使用投影层组合成单个 64 维向量。我们使用 1024 维的

1 Huang, P.-S., He, X., Gao, J., Deng, L., Acero, A., & Heck, L.. Learning Deep Structured Semantic Models for Web Search using Clickthrough Data. ACM International Conference on Information and Knowledge Management (CIKM), 10 (2013).

GoogLeNet 向量作为图像，并使用两个完全连接的层（第一层从 1024 维到 288 维，第二层从 288 维到 64 维）将其进一步缩小到 64 维。产品标题和卖家名称文本也会像查询一样处理，并减少到 64 维。因此，我们最终得到两个 64 维的向量，分别代表查询和产品。此模型训练的目标是将所有点击对<查询，产品>的向量间余弦相似度最大化，反之，此训练过程会采样负对（不相关的对），并对负对的向量余弦相似度做最小化处理。总的来说，这个方法由相关查询和产品详细信息作为引导，从而将 GoogLeNet 向量的维度从 1024 维减少到 64 维。最后，我们将这两个用来减少图像矢量维数的完全连接层（1024→288 和 288→64），用于输入查询图像及构建广告索引。

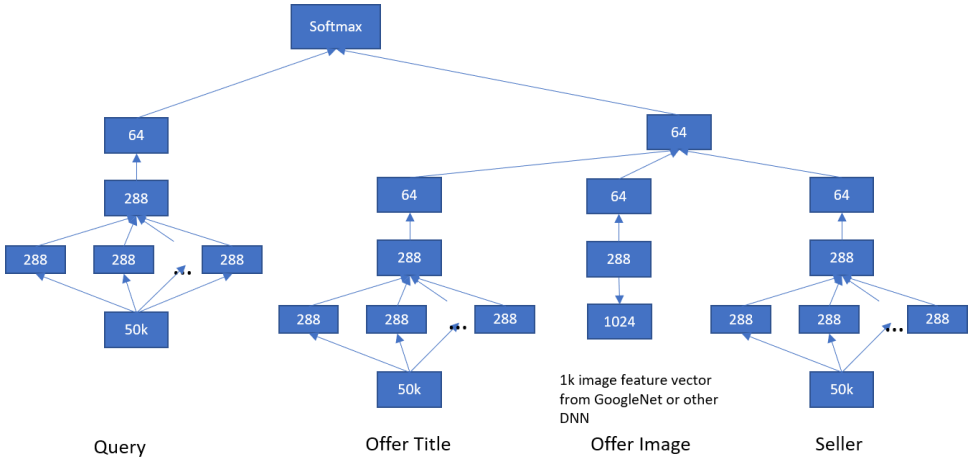


图 7.2　Rich-CDSSM 模型降维

读者可能想知道我们提出的模型是否是最佳的。检验模型是否最佳有两种方法：一种方法是将所有文本信息（查询、产品标题、产品卖方名称）移动到一方，并且仅在另一方保留报价图像。这可以避免图像矢量缩减器的懒惰：如果产品标题 DNN 部分已经产生匹配查询意图的关键信息，那么在训练期间，图像的两个完全连接层可能不会被激励提取相同的信息，以致在使用时可能丢失一些关键信息。另一种方法是在框架中添加更多的产品相关数据，例如价格、品牌、颜色、

性别和年龄组，以进行更精确的降维。

7.2.3 快速最近邻搜索系统

即使向量维数为 64 维，我们也不可能通过线性搜索来推荐如此大的广告语料库。于是我们采用邻域图搜索方法[1]，又称为 NGS，用 NGS 来构建快速检索索引。NGS 离线构建邻域图用于索引广告产品数据点，在线检索期间，它能在邻域图进行本地搜索，用 best-first search 扩展邻域。除此之外，我们还对广告语料库进行分区并构建多个索引，因此每个请求都将广播到分区，以便用并行分布式进行检索。因此，检索有极高的效率，在 99 百分位数下花费的时间不到 5ms。我们用了较小的距离阈值以确保更好的召回率。

7.2.4 精密层

通过 NGS，系统可以快速检索产品广告，其重点是提高相关产品的召回率。在使用 NGS 之后，我们实施了一个精密层来控制推荐质量。

精密层基于机器学习模型，提取图像和产品的特征来学习人类所标注的标签。我们为评委设计了一个指南，根据应该匹配的几个关键点，让评委给每个<查询图像，产品广告>对标记上多尺度标签。举例来说，最重要的是，查询图像中的产品需要与所选广告产品的类别相符。如果产品不匹配，则该对应被视为不良。另外，如果产品类型相符，且品牌、颜色或适用性别也匹配，则该对更具相关性，应得到更好的尺度标签。作为第一步，我们使用二元标签{defect，not-defect} 来训练决策树的模型，并在预估分数低于阈值时将 NGS 选择的广告过滤掉。以下介绍我们所抽取的特征。

首先，我们使用与 NGS 一样的 64 维向量余弦相似度作为最基础的<查询图

1 Wang, J., & Li, S. Query-driven Iterated Neighborhood Graph Search for Large Scale Indexing. Proceedings of the 20th ACM International Conference on Multimedia.New York, NY, USA: ACM,2012:179-188.

像，产品广告> 图像相似特征。然后，运行另一个 GoogLeNet 模型来预测输入图像的产品类别，总共大约有 1000 个产品类别。我们将查询图像的前 3 个预测类别和置信度分数用作特征。查询图像的产品类别和所选广告产品类别是否相符是一个很重要的特征。另外，置信度分数对于抑制不良广告特别有用。当输入图像较为困难时，例如图像模糊、产品图像被裁剪得不完整等，产品类别的置信度得分通常较低，所选择的广告也常不相关，人为标签也是如此。该模型可以将置信度得分与标签相关联，并降低模型得分以进行过滤。请注意，广告的类别预测不仅基于产品图像本身，还基于文本元数据（如标题），因此，产品的类别预测通常比较准确。

在这个模型下，读者可以想象，还有许多特征可以控制产品广告的质量，例如产品主色、适用性别、品牌或 Logo 等。这些图像特征可以用另外事先训练好的模型抓取。同时，使用 OCR 在产品图像上检测到的字词也能与产品广告的文本数据比较。

7.2.5 端到端服务系统

图 7.3 所示为 API 背后的端到端服务系统。为了处理大量传入的查询图像，我们部署了负载均衡的分布式服务系统，称为深度学习推理系统。首先，此系统利用 Rich-CDSSM 模型，将查询图像的 1024 维像素编码（GoogLeNet 隐藏矢量）降低到 64 维。然后，将该向量发送到分布式 NGS 索引以进行 ANN 搜索。每个分区根据编码矢量的距离及精确层模型的分数返回前 K 个最佳结果。精确层与 NGS 索引放在广告相同的分区机器上，以便存取并共享广告的数据。所有分区的结果进一步汇总并重新排序，以选择最佳的视觉相似结果。此处，还会应用基于出价的竞价，以便选择出价高、获利更高的广告。

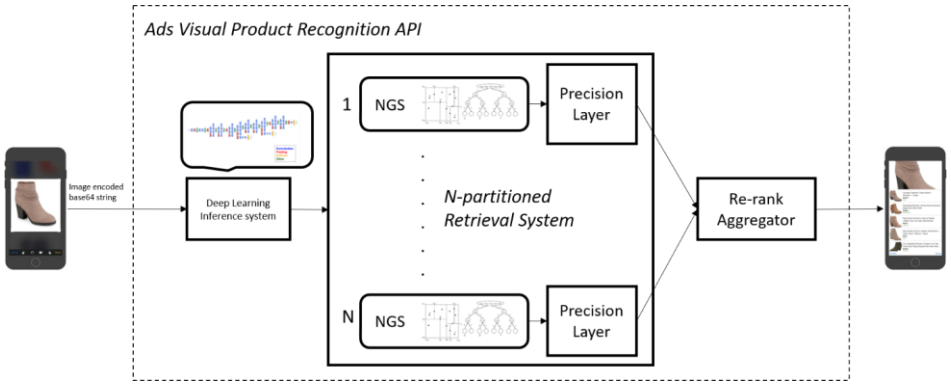


图 7.3　API 背后的端到端服务系统

7.3　评测

下面我们根据产品广告流量评估 API 的精确度。取样 10,000 个在 3 个月内最常展示的广告，代表用户最常见且最关心的产品。产品广告涵盖了 30 个不同根类别和 2048 个叶类别。图 7.4 所示为关于餐桌的广告图像，我们将这些广告图像作为查询图像来访问系统。将每个查询广告的类别、颜色和品牌作为基本事实标准(ground truth)，并将它们与系统返回的前 20 个广告中最受欢迎的类别、颜色和品牌进行比较。Top1 与 Top3 产品的准确率可以在图 7.5 中找到。读者请注意，这里所展示的数字绝对值只能作为参考，但读者可以从数字相对大小的趋势来理解各种方法的优劣，以及不同类型产品有不同的判断难度。

图 7.4　广告图像作为系统评估中的查询图像的示例

Root Category	Root Category @Top-1	Root Category @Top-3	Leaf Category @Top-1	Leaf Category @Top-3	Brand @Top-1	Brand @Top-3	Color @Top-1	Color @Top-3
kitchen housewares	82.83%	90.19%	68.22%	80.72%	46.50%	60.98%	37.18%	55.67%
jewelry watches	78.72%	84.04%	60.64%	73.40%	9.57%	12.77%	14.29%	22.45%
home furnishings	77.61%	91.39%	48.11%	67.62%	14.70%	24.11%	30.08%	43.73%
collectibles memorabilia	77.05%	88.52%	77.05%	86.89%	1.64%	3.28%	66.67%	66.67%
beauty fragrance	71.58%	85.52%	43.43%	57.37%	25.74%	38.07%	7.84%	23.53%
clothing shoes	70.47%	81.09%	57.25%	70.73%	17.88%	24.87%	28.99%	38.76%
cameras optics	69.23%	77.83%	57.01%	67.87%	47.96%	59.28%	53.52%	67.61%
health wellness	69.06%	82.12%	39.35%	55.83%	29.71%	38.48%	20.30%	36.84%
books magazines	67.90%	81.48%	43.21%	53.09%	1.23%	2.47%	0.00%	33.33%
car garage	66.43%	86.43%	39.29%	53.93%	15.71%	30.00%	35.85%	56.60%
computing	63.66%	80.98%	51.95%	66.34%	61.71%	76.34%	37.11%	55.15%
baby nursery	63.16%	68.42%	63.16%	68.42%	57.89%	78.95%	7.69%	7.69%
office products	60.00%	74.29%	47.27%	61.04%	24.16%	31.43%	35.80%	46.91%
pet supplies	59.62%	76.92%	47.12%	67.31%	37.50%	51.44%	33.33%	61.11%
lawn garden	58.74%	74.71%	49.30%	63.05%	25.29%	34.73%	24.73%	41.94%
tools hardware	56.90%	76.31%	40.90%	58.00%	24.66%	33.58%	19.63%	35.56%
sports outdoors	52.49%	62.91%	41.21%	51.41%	29.93%	42.95%	28.19%	44.97%
toys	51.37%	65.88%	43.92%	58.04%	22.35%	32.16%	16.67%	25.00%
All	**61.37%**	**76.38%**	**45.08%**	**60.19%**	**29.09%**	**39.23%**	**29.44%**	**44.73%**

图 7.5　产品类别、品牌和颜色预测的前 N 个的准确率

在图 7.5 中显示，对于厨房和家庭用品（kitchen housewares）、珠宝和手表（jewelry watches）、家居饰品（home furnishings）、服装和鞋类（clothing shoes）等热门类别，该系统在根类别预测方面可达到 82% 的 Top1 准确率，90% 的 Top3 准确率。对于所有类别的产品图像，根类别预测为 61% Top1 准确率和 76% 的 Top3 准确率，叶类别预测具有 45% 的 Top1 准确率和 60% 的 Top3 准确率。对于服装、鞋类、珠宝和手表等时尚产品，叶类别 Top3 的准确率可分别为 70% 和 73%。

视觉产品搜索系统的最大问题主要是查询图像的复杂性，如图 7.4 所示的第一个图像，图上有数个产品对象（桌子和椅子），而且图像的背景是复杂的，这种图像会增加视觉产品搜索的难度。

品牌准确率在不同的产品类别中有所不同，它在计算机类（computing）中具有 61% 的 Top1 准确率，在珠宝和手表中却具有 10% 的 Top1 准确率。预测品牌非常具有挑战性，特别是当品牌标识很小且部分被遮挡时，有时甚至在图像

上用肉眼也无法观察到产品的品牌信息。如 7.2.4 节所谈到的，可以训练独立的模型直接从图像中抽取品牌信息，例如一些品牌名称、Logo 或特殊品牌产品设计。对于具有多种颜色的产品，我们不仅需要专注于它们的第一个主导颜色，而且要将广告商提供的颜色值从“白色”“黑色”“灰色”“红色”等扩展到 RGB 或 HSV 中明确定义的颜色段，以便于判断色彩相近程度。

至于 Rich-CDSSM 模型，我们使用 5500 万个点击<查询，产品>对训练模型。对于精密层中的相关性模型，使用 5000 个标记对训练模型，并交叉验证 AUC 来评估。若以 NGS 一样的 64 维向量余弦相似度作为单一特征，AUC 得分为 0.73；而使用所有特征来训练决策树的模型，AVC 得分为 0.83。

通过发送 1000 个图像请求，并计算平均值来测量 API 延迟。API 的平均延迟时间为 283ms，只比一般文本查询请求慢约 5%。

7.4 用于演示的 Visual Shopping Assistant 应用程序

下面我们使用应用程序来演示 API 的功能。在 iOS 平台上编写一个应用程序 Visual Shopping Assistant，此应用程序允许用户从图像中搜索视觉上相似的产品广告，并通过单击其中一个广告完成购买，如图 7.6 所示。在后端，应用程序直接在手机上将图像像素作为 base64 编码的字符串发送到 API，并收到产品广告作为回报。

用户可以使用相机拍摄感兴趣的产品照片（见图 7.6 左上）。然后，应用会将照片像素发送到 API 以获取产品广告列表，并在拍摄的照片下方垂直显示（见图 7.6 右上）。用户还可以裁剪照片的一部分进行搜索（见图 7.6 左下），API 将仅返回与裁剪照片相关的产品广告（见图 7.6 右下）。

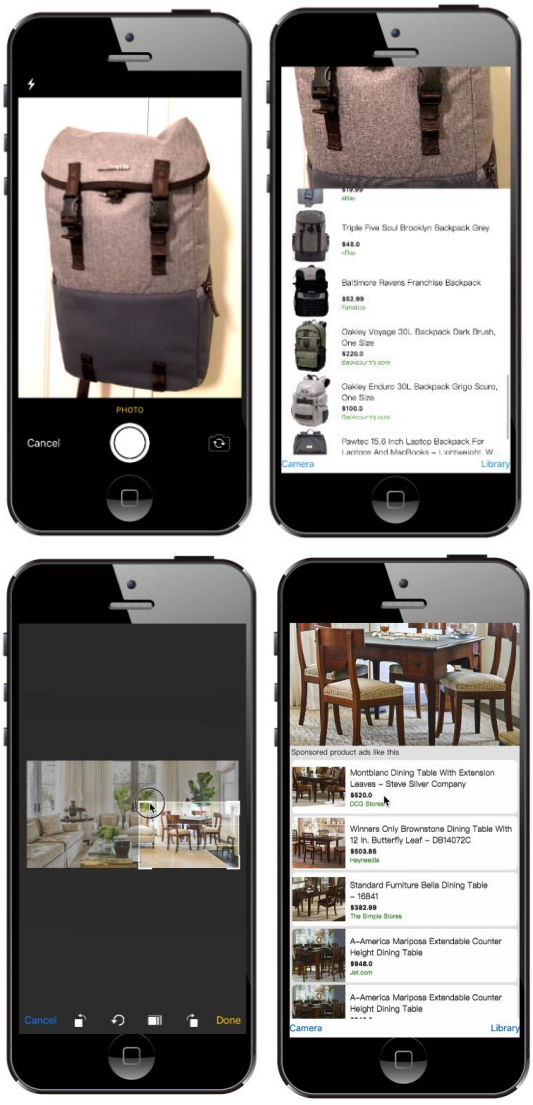


图 7.6　iOS 应用程序 Visual Shopping Assistant

7.5　相关工作

我们基于低维的图像编码向量，建立快速 ANN 搜索机制，来实现以图搜图。

利用 Rich- CDSSM 模型来减少 GoogLeNet 1024 维编码向量的维度，使用产品广告的文本元数据和搜索查询作为指南。

Bing 搜索团队也进行了降维，但他们使用 PCA（主成分分析）直接在图像编码向量上进行降维。我们的 ANN 选择方法不同于 Bing 搜索团队和 Pinterest 的传统信息检索方法。Bing 使用“视觉词”建构索引：GoogLeNet 1024 维向量到 PCA 输出的 100 个维度中的每 4 个维度，Bing 搜索团队将每 4 个浮点数聚类为 256 个聚类，聚类的 ID 则可视为单词，即“视觉词”。eBay 团队将图像向量直接转换为二进制文件，以便根据汉明距离进行快速检索。

在所有视觉搜索产品（Pinterest、eBay、Bing 搜索、阿里巴巴）中，都有一个准确率和一个排名层来优化所选结果。诸多系统也都是通过提取特征来训练模型以学习人为标签或用户点击行为的。而抽取的特征也类似：包括抽取的类别和颜色、预测的置信度，以及诸如产品标题或图像标题的文本元数据。

7.6 本章小结

视觉搜索（以图搜图）已成为 IT 行业中一个投入巨资的领域。在本章中，我们展示了一个视觉产品识别与搜索 API，以便从第三方获取图片并回传产品推荐。我们讨论了 API 背后的技术栈，使用先进的图像 DNN 编码器进行图像理解，以及快速检索的 ANN 搜索方法。

本章还介绍了此 API 背后的一个实现个案：利用 GoogLeNet DNN 将图像编码为浮点矢量，并且利用 CDSSM 架构将编码矢量降维进而加速 ANN 搜索的工作，最后也介绍了一个用于提高广告质量的精密层。

Deep Learning Model and Application

第 8 章

Seq2Seq 模型在聊天机器人中的应用

8.1 Seq2Seq 模型应用背景

本章我们将详细介绍 Seq2Seq（序列到序列）模型，以及其在聊天机器人中的应用。Seq2Seq 模型是由 RNN 延伸出来的被广泛使用的深度学习模型，它在机器翻译方面有着非常优异的表现。[1]举例来说，Seq2Seq 模型能将一句法文翻译成英文：只要将大量相对应的<法文句，英文句>数据集提供给 Seq2Seq 模型进行训练，Seq2Seq 模型便能学习法文和英文之间的关系，以及法文和英文语法的连接等。最终训练好的模型便能将法文翻译成英文。Seq2seq 模型比传统的翻译模型有更高 BLEU[2]得分。

在自然语言领域里，Seq2seq 模型除了在机器翻译方面的应用，还可以用来训练并学习任何类型的配对序列数据。Vinyals 曾经用客服中心的对话日志来训练一个 Seq2Seq 模型[3]，希望能训练出自动回答客户问题的自动客服机器人。举例来说，客服对话日志中包含着<客户问题，服务人员回答>的大量配对数据，用它来训练 Seq2Seq 模型，便可以使它读取新的用户问题并回复“机器翻译”的答案。Kannan 采用电子邮件回复日志来构建一个模型[4]，当用户在手机上读邮件时，Seq2Seq 模型能根据邮件内容建议多个回复样本以供用户选择，以减少使用者的打字时间，提高便利性。

---

1 Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Advances in neural information processing systems, 2014:3104–3112.

2 BLEU（Bilingual Evaluation Understudy Score）是一个用来评断生成句子与标准句子的相似度的评分标准。若生成句子和标准句子完全一样，BLEU 得分为 1.0。反之，完全不匹配的生成句子得分为 0。

3 Vinyals, O., & Le, Q. V. A Neural Conversational Model. CoRR, (2015).

4 Kannan, A., Kurach, K., Ravi, S., Kaufman, T., Miklos, B., Corrado, G., . . . Ramavajjala, V. Smart Reply: Automated Response Suggestion for Email. Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD). 2016.

8.2 Seq2Seq 模型的应用方法

下面介绍 3 个基于 Seq2Seq 模型的应用方法。

生成句子：回答问题的聊天机器人需要一个知识库来搜索答案，所以我们利用现有推荐系统作为搜索知识库，以便寻找最适合的答案。我们应用 Seq2seq 模型来理解用户问题，并将问题重写为一个普通的推荐系统能理解的标准搜索查询句(query)。重写的标准搜索查询被递交至推荐系统检索出一系列的候选答案。除了基本的 Seq2Seq 模型，还利用注意力机制（attention）来提高改写的质量。在 6.3 节将详细介绍 Seq2Seq 模型与注意力机制的运作方式，而且注意力机制可以使该模型重写问题至更高质量的搜索查询句（更高的 BLEU 得分、具有更高的人类标记的质量）。将模型对用户查询进行重写后的语句（以下简称重写）递交给一个商业搜索引擎，可以直接从搜索引擎中检索出广告，证明该系统具有显著的商业价值。

概率估计：除生成句子外，Seq2Seq 模型的另一个强大功能是其概率估计的统计特性。简单来说，Seq2Seq 模型既可以根据句子对的数据集来生成句子，也能为一个句子对进行打分。也就是说，当以句子 A 输入给模型时，Seq2Seq 模型能算出由句子 A 生成句子 B 的概率是多少。当用作评估工具时，Seq2Seq 模型的概率估计模块能计算得到后验概率，可以作为推荐候选答案的评价标准。我们可以使用它重新打分并从推荐系统返回的候选集精选出更佳的结果。

信息导向框架与聊天机器人：这是本章的重点，将 Seq2Seq 模型当成一个概率估计模型用于信息导向评估和用户互动的中心模型框架。Seq2Seq 模型能在人机互动的过程中引导着一个像聊天机器人这样的代理人（agent）。聊天机器人一个很重要的任务是尝试在交互会话中识别用户的意图。我们会描述代理人如何使用 Seq2seq 模型估算器来计算条件熵，并通过 Naives Bayes 程序，在每次新的使用者信息到来时更新此条件熵。代理人迭代使用该信息做出以下决定：要么直接提出建议，要么在不确定用户意图时提出进一步的问题来收集更多的信息。

8.6.7 节将介绍使用这个框架构建的一个聊天机器人应用程序。应用程序是建立在能推荐产品的商业搜索引擎上。如果用户询问一个带有产品意图的问题，那么聊天机器人会推荐一个产品广告。当用户意图不清楚时，它能积极地询问用户种种根据产品属性制定的问题，以达到最大化预期信息增益。

接下来我们将详细介绍 Seq2Seq 模型的细节。

8.3 含有注意力机制的多层 Seq2Seq 模型

我们的模型是含有注意力机制（attention）的 Seq2Seq 模型，如图 8.1 所示。Seq2Seq 模型包含了编码器和解码器，且序列中的每个节点都由多个 LSTM 层堆叠而成。若给定一个句子对<源句子 A，目标句子 B>，编码器先将源句子 A 读入，而解码器根据源句子 A 的信息试着生成新的句子来仿效目标句子 B。下面我们对模型进行详细的解释。

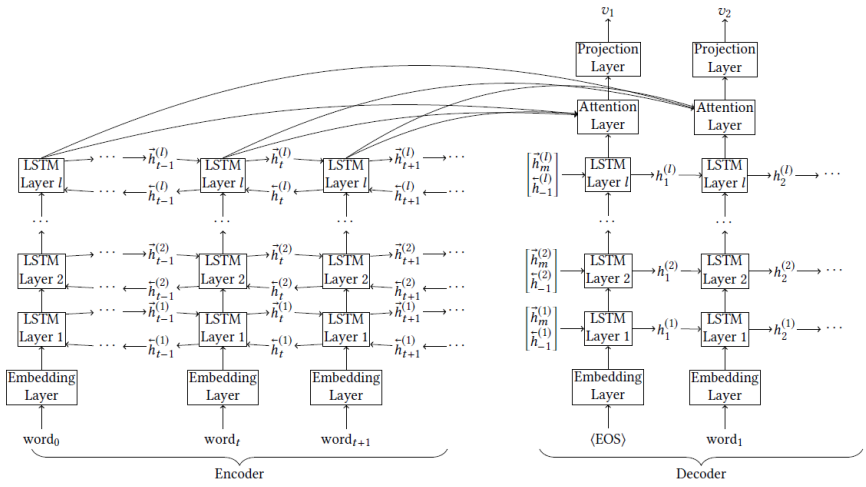


图 8.1 双向多层 LSTM 编码器+ LSTM 注意力机制解码器

8.3.1 词嵌入层

词嵌入层（Word embedding layer）将每一个单词转换为矢量表示，该层需

要矩阵参数$\boldsymbol{W}_{\text{emb}} \in \mathbf{R}^{d_{\text{emb}} \times |V|}$。当一个带有索引 i 的单词输入时，词嵌入层将其转化成$\boldsymbol{W}_{\cdot,i}$，即为矩阵的第 i 列，维度为d_{emb}的向量。编码器和解码器有各自的嵌入层，各自有独立的参数需要训练，即两个$\boldsymbol{W}_{\text{emb}}$矩阵。

8.3.2 可变深度 LSTM 递归层

具有深度 L 的 LSTM 递归层由 L 个垂直堆叠的 LSTM 块组成。每个 LSTM 块有 3 个输入：$\boldsymbol{e}_t$、$\boldsymbol{c}_{t-1}$和$\boldsymbol{h}_{t-1}$，其中$\boldsymbol{e}_t$代表下面一层的输入，$\boldsymbol{c}_{t-1}$和$\boldsymbol{h}_{t-1}$是当前层上一步（t-1 时间点）的输入。它的输出$\boldsymbol{h}_t$按以下方式计算：

$$
\begin{aligned}
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_{ei}\boldsymbol{e}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_{ef}\boldsymbol{e}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \cdot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \cdot \tanh(\boldsymbol{W}_{ec}\boldsymbol{e}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_{eo}\boldsymbol{e}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \cdot \tanh(\boldsymbol{c}_t)
\end{aligned}
$$

其中·符号表示向量之间的元素乘积。LSTM 是一种增强型的递归神经网络（RNN），通过维持额外的记忆单元矢量$\boldsymbol{c}_t$，并引入输入门$\boldsymbol{i}_t$、忘记门$\boldsymbol{f}_t$和输出门$\boldsymbol{o}_t$来解决 RNN 的短期记忆问题。

关于 LSTM 优点的详细讨论，由于篇幅限制，本书对此不再进行详细讨论，读者可查阅相关资料。[1]对于最低的 LSTM 层，$\boldsymbol{e}_t$是嵌入层的输出，其维度为d_{emb}，因此$\boldsymbol{W}_{e*} \in \mathbf{R}^{d_h \times d_{\text{emb}}}$、$\boldsymbol{W}_{h*} \in \mathbf{R}^{d_h \times d_h}$和$\boldsymbol{b}_* \in \mathbf{R}^{d_h}$是要训练的参数。对于以上的 LSTM 层，$\boldsymbol{W}_{e*}$、$\boldsymbol{W}_{h*} \in \mathbf{R}^{d_h \times d_h}$和$\boldsymbol{b}_* \in \mathbf{R}^{d_h}$是要训练的参数。这里的$\sigma()$指的是 sigmoid 非线性激活函数。对于编码器来说，每个 LSTM 块实际上是双向的（BLSTM），前向 LSTM 和后向 LSTM 各自输出的隐藏向量将被串联起来输入到堆栈层的

1 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation 9, 8 (1997), 1735–1780.

LSTM 块。而输入到解码器的最终向量是两个方向 LSTM 的最后一个隐藏向量所串联起来得到的：$[\overrightarrow{\boldsymbol{h}_m};\overleftarrow{\boldsymbol{h}_{-1}}]$。然而不同于编码器所使用的双向 LSTM，每个解码器的 LSTM 块仅具有前向方向。因此解码器中的$\boldsymbol{d_h}$应该是编码器$\boldsymbol{d_h}$大小的两倍；解码器的 BLSTM 层有相同维度的输出向量。

8.3.3 注意力机制层

对于解码器的每个顶部隐藏矢量$\boldsymbol{h_t}$，我们另外算出一个矢量$\mathbf{g_t}$与其串联以得到更好的效果。它是通过来自编码器的顶部隐藏矢量$\boldsymbol{s_i}$加权平均组合获得的。有关注意力机制的相关知识将在后面的【 知识点讲解 】中介绍。矢量$\boldsymbol{g_t}$与顶部 LSTM 层$\boldsymbol{h_t}$的输出矢量串联后，我们使用了一个完全连接层将尺寸缩小回与输入隐藏矢量相同的尺寸：$\widehat{\mathbf{h}_t} = \mathrm{ReLU}(\boldsymbol{W}_c[\boldsymbol{g}_t;\boldsymbol{h}_t] + \boldsymbol{b}_c)$，其中 ReLU 是非线性激活函数：$\max(0,\cdot)$。这里需要训练的参数是$\boldsymbol{W}_c \in \mathbf{R}^{d_h \times 2d_h}$和$\boldsymbol{b}_c \in \mathbf{R}^{d_h}$。输出的$\widehat{\boldsymbol{h}_t}$将传递到下一个投影层。

8.3.4 投影层

投影层（projection layer）是 Seq2seq 模型的关键最终步骤，它将隐藏向量转化成单词的概率分布，而概率最高的字则被视为解码器在时间 t 的翻译结果。目标是希望时间 t 的翻译结果和目标序列的第 t 个字相同。

精确地说，投影层将组合的隐藏矢量作为输入，并输出维度$|V|$的矢量。其参数包括权重矩阵 $\boldsymbol{W}_p \in \mathbf{R}^{|V| \times d_h}$ 和 $\boldsymbol{b}_p \in \mathbf{R}^{|V|}$。步骤 t 的输出计算为 $\boldsymbol{V}_t = \mathrm{softmax}(\boldsymbol{W}_p\widehat{\boldsymbol{h}_t} + \boldsymbol{b}_p)$。注意，经过了 softmax[1]函数，$\boldsymbol{V}$是非负向量，其总和为 1，因此可以将其视为词汇表$\boldsymbol{V}$上的概率分布。

索引w_t的单词在第t步的概率可以利用 seq2seq 模型得到，缩写为：

$$v_t(w_t) \tag{1}$$

1 $\mathrm{softmax}_{ij}(x) = \frac{\exp(x_{ij})}{\sum_k \exp(x_{ij})}$

8.3.5 损失函数（loss function）和端到端训练

下面我们对 Seq2seq 模型进行端到端训练，共同学习所有上述参数。而学习的目标如 8.3.4 节所说，是希望每一个时间 t 的翻译结果和目标序列的第 t 个字相同。我们希望模型能够逐渐减少翻译误差，也就是对数据集的所有句子对<源句子 A,目标句子 B>的翻译误差总和进行最小化训练。训练时，目标句子的第 $t-1$ 个字是解码器第 t 个字嵌入层的输入。

我们用交叉熵（cross entropy）来量化误差。训练集中的每一对为源句子 Src 和目标句子 Tgt，其中$\mathbf{Tgt} = w_{t_1} \cdots w_{t_n}$，通过编码器对 Src 进行编码，该对的损失量是$v_t$和目标句子 Tgt 中每个单词交叉熵损失的总和。也就是说，目标是让投影层在每个时间点都能让w_t的概率最大化。

$$\mathrm{loss} = -\sum_{\mathrm{<Src,Tgt>}} \sum_{t \in \mathrm{Tgt}} \log v_t(w_t)$$

【知识点讲解】注意力机制

为了更进一步地提升 seq2seq 模型的性能，不同的机制相继被提出，其中包括注意力机制。Loung[1]在解码器顶部的隐藏向量增加了编码器隐藏向量的加权平均。通过添加注意力机制至 Seq2Seq 模型中，能够让输入和输出字符串对齐，BLEU 得分为 5.0。

注意力机制旨在解决对递归神经网络的短期记忆问题。即使 LSTM 模型和 GRU 网络设计为了具有较长的记忆性，也很容易丢失很久以前所读入的信息（源序列最早读入的字）。注意力机制背后的直观解释是，在序列解码过程的每一步，我们强制神经网络再次回顾源序列的每个字所对应的隐藏向量，并将有用的信息

1 Luong, M.-T., Pham, H., & Manning, C. D. Effective Approaches to Attention-based Neural Machine Translation. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing . Lisbon: Association for Computational Linguistics.9(2015), 1412-1421.

留下用以增强当前使用的隐藏向量。

为方便起见，我们使用了一些符号。让源序列的顶部隐藏向量为$\boldsymbol{s}_1 \cdots \boldsymbol{s}_m$，而当前目标词的顶部隐藏向量为 $\boldsymbol{h}_i$。以下总结了 4 种不同的注意力机制（dot、general、concat、tensor），目的皆在计算当前隐藏向量 $\boldsymbol{h}_i$与源隐藏向量$\boldsymbol{s}_j$的相似性[1,2]，而所算出的相似度可用来算出源隐藏向量$\boldsymbol{s}_1 \cdots \boldsymbol{s}_m$的加权平均值。重要且相关的源序列单词应该具有更大的权重，因此它具有翻译对齐的功效，并称为“注意”层。对于这 4 种注意力机制，可以首先分别计算目标词 i 的权重向量 $\boldsymbol{a}_i$：

$$\text{(dot)} \quad \tilde{\boldsymbol{a}}_{ij} = \boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{h}_i$$

$$\text{(general)} \quad \tilde{\boldsymbol{a}}_{ij} = \boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{W}_g \boldsymbol{h}_i$$

$$\text{(concat)} \quad \tilde{\boldsymbol{a}}_{ij} = \boldsymbol{W}_{cc}[\boldsymbol{s}_j; \boldsymbol{h}_i]$$

$$\text{(tensor)} \quad \tilde{\boldsymbol{a}}_{ij} = U\left(\boldsymbol{s}_j^{\mathrm{T}} \boldsymbol{W}_g \boldsymbol{h}_i + \boldsymbol{V}[\boldsymbol{s}_j; \boldsymbol{h}_i]\right) + \boldsymbol{b})$$

然后用$\boldsymbol{a}_{i,\cdot} = \mathrm{softmax}(\tilde{\boldsymbol{a}}_{i\cdot})$，通过$\boldsymbol{g}_i = \sum_{j=1}^{m} \boldsymbol{a}_{ij} \boldsymbol{s}_j$获得注意向量，将它与$\boldsymbol{h}_i$组合并串联，作为输入传递给投影层。在此可训练的参数为$\boldsymbol{W} \in \mathbf{R}^{d_h \times k \times d_h}$、$\boldsymbol{V} \in \mathbf{R}^{k \times 2d_h}$和$\boldsymbol{b} \in \mathbf{R}^k$。

4 种注意力机制针对的目标不同。虽然 dot 和 general 旨在发现源和目标之间的相似性，但最后一种注意图机制更侧重于单词之间的非线性相互作用。[3]

---

1 Minh-.ang Luong, Hieu Pham, and Christopher D. Manning. E.ective Approaches to A.ention-based Neural Machine Translation. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Lisbon, Portugal, 2015:1412–1421.

2 Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning With Neural Tensor Networks for Knowledge Base Completion. In Advances in Neural Information Processing Systems 26,C. J. C. Burges, L. Bo.ou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds). Curran Associates, Inc., 2013:926–934.

3 Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning,Andrew Y Ng, and Christopher Po.s. Recursive deep models for semantic compositionality over a sentiment treebank. In Proceedings of the conference on empirical methods in natural language processing (EMNLP), Vol. 1631,2013.

【知识点讲解】概率估计

在给定源序列 Src 的情况下，上述 Seq2Seq 模型能够给出目标序列$\mathrm{Tgt} = w_{t_1} \cdots w_{t_n}$出现的概率估计。首先，根据条件概率的链式规则，我们有：

$$\Pr(\mathrm{Tgt}|\mathrm{Src}) = \Pr(w_{t_1}|\mathrm{Src}) \cdot \cdots \cdot \Pr(w_{t_n}|w_{t_1} \cdots w_{t_{n-1}}, \mathrm{Src})$$

对于目标序列中的第 i 个字w_{t_i}，条件分布$\Pr(w_{t_i}|w_{t_1} \cdots w_{t_{i-1}}, \mathrm{Src})$可以从 Seq2Seq 模型估算，即：

$$\widehat{\Pr}(w_{t_i}|w_{t_1} \cdots w_{t_{i-1}}, \mathrm{Src}) = v_i(w_{t_i})$$

将链式规则步骤与 Seq2Seq 模型估计器相结合，我们得到估计目标序列概率，即：

$$\Pr(\mathrm{Tgt}|\mathrm{Src}) = \prod_{i=1}^{n} v_i(w_{t_i}) \quad (2)$$

8.4 信息导向的自适应序列采样

在上述内容中，我们着重于介绍 Seq2Seq 模型和注意力机制，以及其句子生成能力与概率估计能力。然而，目前所讨论的只局限于一次性且孤立的事件，例如根据源序列来估算目标序列概率，但在聊天机器人、虚拟代理或交互式网页等新互动渠道中，这样的模型是不够的。

聊天机器人，必须具有说话的适应性，并能向用户提出问题以达到其真实的目标。所谓的自适应，指的是它能对使用者当前的状态做理解与估计，且它能够根据当前的状态来进行动态采样。所谓动态采样，即是对使用者提出有效的问题以尽快达到使用者的需求。换句话说，它应该解释用户意图并引导用户回答问题，以最有效的方式实现使用者的目的。

接下来，我们将介绍如何集成 Seq2Seq 模型来进行估算，确定下一个采样方向，并在代理人有信心时提出建议。

8.5 多轮项目推荐

考虑一个我们想提出建议的场景。将 π 表示为所有可能项目集的先验分布。在此设置中，每个项目都可以表示为序列，例如产品的标题。现在，假设我们已有来自用户的k个输入序Input_1^k，例如，与试用者的k轮互动后，项目集上的后验分布应相应地改变，反映了用户已提供多项信息的事实。应用贝叶斯规则，我们有：

$$\Pr(\text{Item}|\text{Input}_1^k) = \frac{\pi(\text{Item})\Pr(\text{Input}_1^k|\text{Item})}{\sum_{\text{Item}}\pi(\text{Item})\Pr(\text{Input}_1^k|\text{Item})}$$

在朴素贝叶斯（naive Bayesian）框架下，我们假设条件独立：

$$\Pr(\text{Input}_1^k|\text{Item}) = \prod_{i=1}^{k}\Pr(\text{Input}_i|\text{Item})$$

结合上述两个公式，更新规则就变成了：

$$\Pr(\text{Item}|\text{Input}_1^k) = \frac{\pi(\text{Item})\prod_{i=1}^{k}\Pr(\text{Input}_i|\text{Item})}{\sum_{\text{Item}}\pi(\text{Item})\prod_{i=1}^{k}\Pr(\text{Input}_i|\text{Item})} \quad (3)$$

我们注意到，每个概率项$\Pr(\text{Input}_i|\text{Item})$由公式（2）中的 Seq2Seq 模型的概率估计给出。

8.6 熵作为信心的度量

8.6.1 直观的定义和讨论

熵是概率分布的函数，它测量分布的随机性，即不确定性。我们使用它来确定代理人的信心，或者指代理人对使用者理解的模糊程度。它源于信息理论，量化了 IID 随机源序列的可压缩性[1]，其后来被广泛应用于其他领域，包括计算机视

1 Thomas M Cover and Joy A .omas. Elements of information theory. John Wiley & Sons, 2012.

觉和语音识别。例如，Hoch 和 Skilling[1] [2]首先提出的最大熵原理，并已经在图像重建和去模糊方面取得了极大的成功。最大熵原理也在语音识别领域得到应用[3]。在 NLP 中的语言模型，有时也围绕这个想法建立[4] [5]。下面我们给出熵函数的正式定义。请注意，在下面的条件熵定义中，它不是在它所调节的随机变量上进行平均的，因此它本身就是一个随机变量。

定义（熵，条件熵），给出一对离散的随机变量(X,Y)，其中X取x的值，Y取y的值。表示它们的联合分布为$P_{X,Y}(x,y)$和边际概率$P_X(x), P_Y(y)$。

（1）X的熵定义为：

$$H(X) = -\sum_{x\in \mathrm{X}} P_X(x)\log P_X(x)$$

（2）给定 $Y=y$ 的 X的条件熵是：

$$H(X|Y=y) = -\sum_{x\in \mathrm{X}} P_{X|Y}(x|y)\log P_{X|Y}(x|y)$$

最后，我们使用$H(X|Y=y) = -\sum_{x\in \mathrm{X}} P_{X|Y}(x|y)\log P_{X|Y}(x|y)$来表示给定$Y$的$X$的预期条件熵。

通常，大的熵值意味着分布更广泛。例如，当熵最大化时，X均匀分布。相反，当熵越小时，分布越集中。$H(X)$实际上意味着分布是否有确定性。

1 John Skilling and RK Bryan. Maximum entropy image reconstruction:general algorithm. Monthly notices of the royal stronomical society 211, 1 (1984),111–124.

2 Jeffrey C Hoch and Alan S Stern. Maximum entropy reconstruction. eMagRes,1996.

3 Jochen Peters. Speech recognition system, training arrangement and method of calculating iteration values for free parameters of a maximum-entropy speech model. 7 (2006). US Patent 7,010,486.

4 Sanjeev Khudanpur and Jun Wu. A maximum entropy language model integrating n-grams and topic dependencies for conversational speech recognition.

5 In Acoustics, Speech, and Signal Processing, 1999 IEEE International Conference on, Vol. 1, 1999:553–556.

8.6.2 序列后验估计的不确定性

条件熵可以作为估计序列后验分布的不确定性度量。请记住，在公式（3）中，我们讨论了当k个用户输入Input_1^k被观察时的后验更新过程。我们将后验不确定性视为这种条件分布的熵，即$H(\text{Item}|\text{Input}_1^k)$。大的后验不确定性则意味着估计是模糊的，因此在做出决定之前需要更多的观察；另一方面，接近于 0 的后验不确定性表明估计已经大大收敛到其最优结果，在这种情况下，可以做出确定的建议。接下来，我们将解释如何对更多观察结果进行抽样或确定最佳问题，以确定它是否不确定。

8.6.3 信息导向的抽样：最大化预期信息增益的原则

现在，假设代理人能够主动与用户互动，能够向用户询问问题并期望从用户那里得到答案。首先，假设有一组可能的问题，$Q = \{\text{Qst}_1, \ldots, \text{Qst}_q\}$。遵循最大化信息增益原则，我们提出信息导向的抽样算法（见图 8.2）。我们想指出的是，在每个步骤中，最大化预期信息增益的信息导向抽样算法实际上是一种贪婪的降低不确定性的算法。观察结果为：在步骤 n 中提出的信息增益最大化Qst也是步骤 n 中的不确定性最小化器。

下面进行证明。展开公式：

$$I(\text{Qst}; \text{Item}|\text{Input}_1^n) = H(\text{Item}|\text{Input}_1^n) - I(\text{Item}|\text{Qst}, \text{Input}_1^n)$$

注意，$H(\text{Item}|\text{Input}_1^n)$不依赖于$\text{Qst}$，因此，$I(\text{Qst}; \text{Item}|\text{Input}_1^n)$的最大化立即成为$I(\text{Item}|\text{Qst}, \text{Input}_1^n)$的最小化器，反之亦然。

有关聊天机器人应用和问题制定程序的讨论将在 8.6.7 节中讨论。

信息导向的抽样算法

1: **for** $n = 1, 2, \cdots$ **do**
2: The n-th sequence $Input_n$ is collected from the user.
3: Estimate the likelihood, $\Pr(Input_n|Item)$, using the seq2seq likelihood estimator.
4: Update the posterior distribution

$$\Pr(\text{Item}|\text{Input}_1^n) = \frac{\pi(\text{Item})\prod_{i=1}^{n}\Pr(\text{Input}_i|\text{Item})}{\sum_{\text{Item}}\pi(\text{Item})\prod_{i=1}^{k}\Pr(\text{Input}_i|\text{Item})}$$

5: Calculate the conditional entropy $H(\text{Item}|\text{Input}_1^n)$
6: **if** $H(\text{Item}|\text{Input}_1^n) < T$ **then**
7: Return $\arg\max \Pr(\text{Item}|\text{Input}_1^n)$, the most likely item.
8: **else**
9: Choose Qst that maximizes $I(\text{Qst Item}|\text{Input}_1^n)$
10: Propose Qst to user; wait for user feedback Input_{n+1}
11: **end if**
12: **end for**

图 8.2 信息导向的抽样算法

8.6.4 Seq2Seq 模型的 3 个应用程序

根据以上介绍的理论基础，我们建构了 3 个应用程序，如图 8.3 所示。第三个应用程序为完整的聊天机器人，基建构于第一个和第二个应用程序的基础之上。

首先，使用 Seq2Seq 模型建模，将用户问题重写为标准搜索查询句，将其提交至一般的推荐系统。然后，评估基于 Seq2Seq 模型的相关性评分。最后，构建一个能够进行聊天的原型，它能与用户进行互动，可以提出问题以达到最大化信息增益。

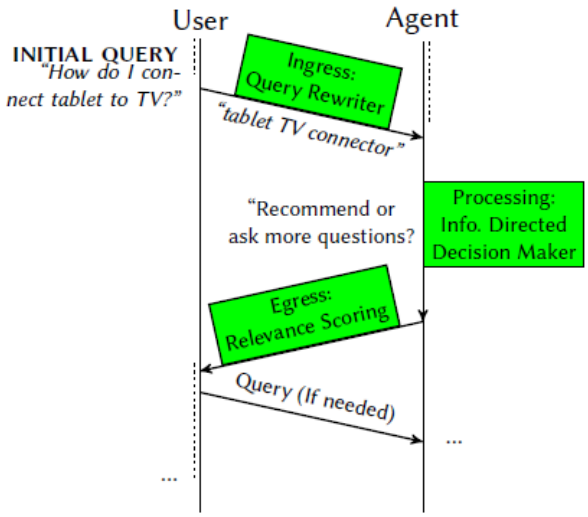


图 8.3　Seq2Seq 模型的 3 个应用程序(标绿色)

8.6.5　应用程序 1：查询理解和重写

查询理解和重写是现代推荐和信息检索系统的重要预处理步骤。在第一个应用程序中，我们应用 Seq2Seq 模型进行问题重写，以最终实现聊天机器人的应用程序。

1. What and How（什么和如何）：问题理解

搜索引擎中关于产品推荐的一个难点是，它不能很好地处理以问题形式出现的查询。这些查询通常是模糊的，隐含地提到产品，并且通常以其他实体的关系形式来表达。例如，用户可能在搜索框中输入类似的问题：

“如何将平板电脑连接到电视？”

从人的角度来看，这种形式的问题可能是想查询一个相关产品：Micro HDMI 线缆。但是，这对信息检索系统提出了挑战，因为查询中不存在与正确的产品相关的明确关键字。而 Seq2Seq 模型可以学习如何重写这样的问题，以使搜索引擎推荐正确的产品。

2. 训练和数据

下面我们训练 Seq2Seq 模型从问题重写至标准查询。为了达到这个目的，我们使用搜索引擎上的“相关搜索”功能所收集的数据集训练模型。当使用者在搜索框中输入特定查询时，相关搜索功能向用户推荐相似的查询列表。我们只保留“what”和“how”开头的用户输入问题，将其当成源序列，并将用户点击的相关搜索查询视为目标序列。用户的点击行为，可以确认推荐的标准查询确实与用户输入的问题相关，也保证了数据集的质量。通过这样做，我们能够收集 1200 万个序列对（问题形式查询，标准查询）的数据集。为了进一步关注最终会引发产品推荐的问题，我们过滤数据集，只保留标准查询本身能有产品推荐的数据。如此一来，共收集了 78.2 万对此类训练样本。表 8.1 总结了训练数据集的统计数据。

表 8.1　查询重写训练数据集的统计数据

	数量（万）	词汇量（万）	平均长度（个字符）	点击数（万）
问题形式查询	31.6	12.6	5.5	78.2
标准查询	48.1	87.0	2.8	78.2

3. 模型细节

下面我们看一下模型细节：编码器和解码器的词汇量大小同设为 $|V|$ = 100k，而不在词汇表 V 里的字被分配了符号<UNK>。我们选择嵌入维度d_{emb}= 100，在解码器端使用隐藏矢量维度为 d_h= 300 的 3 层 LSTM 模型，并且实现了第 6.3.5 节中 4 种不同的注意力机制，后续内容将比较 4 种不同注意力机制的结果。我们使用 Theano[1]实现 Seq2Seq 模型，并在 Tesla K20 GPU 上进行模型训练，以学习 6.3 节中描述的所有参数，训练量总共 10 轮（epoch）。

1 Theano Development Team..eano: A Python framework for fast computation of mathematical expressions. arXiv e-prints abs/1605.02688,5 (2016)

4. 结果和评估

从实验中我们发现了 Seq2Seq 模型的重写在两种方式下有帮助。

首先，虽然许多原始查询与产品相关，但由于查询的形式或其隐含性，它们不会触发产品推荐。在重写之后，它们变得更像关键字并能触发产品推荐，例如：

How to connect my tablet to TV → tablet tv connector

How to repair my broken iphone screen → iphone screen replacement

How to charge my iphone → iphone charger

How to protect my iphone screen → iphone screen protector

其次，重写也有助于检索正确的产品，尤其是当查询中存在隐式或复杂关系时。比如，查询“How to wire car radio”，从人类理解的角度来看，这表明用户已经拥有无线电并正在寻找布线产品。以原始问题形式提交时，系统会推荐许多汽车收音机上的产品。但在 Seq2Seq 模型中将其重写为“radio wiring”后，检索便推荐了正确的产品：无线电线束。再比如，查询“How to fix gps in car”，其原始问题触发了有关移动 GPS 的产品推荐，而在重写之后，返回正确的产品推荐:GPS 支架。

为了定量地评估效果，我们测试不同的（问题表单查询，标准查询）测试数据集。使用 8.6.5-2 节中描述的相同程序收集测试数据集，但是在不同时间段从日志中采样。此外，训练数据集中出现的任何对都已从测试数据集中删除。表 8.2 中总结了测试数据集的统计数据。

表 8.2 查询重写测试数据集的统计数据

问题数量（万）	标准查询数量（万）	对数（万）
3.4	4.2	4.5

5. 重写质量

对于测试数据集中的每个数据对，可以使用不同的 Seq2Seq 模型变体生成重写。我们借用相同的技术来评估查询问题重写的质量，并以数据对中的标准查询作为基线。BLEU 分数常用于评估 Seq2Seq 模型的翻译质量。

表 8.3 中总结了各模型的 BLEU 得分。可以清楚地看到，使用注意力机制的模型始终优于没有注意力机制的基础模型。这种观察不同于最近几次尝试应用 Seq2Seq 模型进行问答的研究结论：注意力机制对相关研究没有帮助。[1]如果我们基于上面的示例进行反思，可能的解释是：问题重写和机器翻译有着相同的源至目标序列的对齐特性，而问题至答案并没有类似的特性。在注意力机制中，general 注意力机制表现最好，而 concat 注意力机制表现最差。严格来说，concat 注意力机制的数学式不直接将源和目标词的隐藏向量进行对齐。另一方面，dot 和 general 的注意力机制直接对齐。最后，tensor 注意力机制排名第二，仅比 general 注意力机制差一点。我们怀疑它的结构或许太过复杂，不容易训练。

表 8.3　在不同模型下，问题形式查询与标准查询的 BLEU 得分

模　　型	BLEU 分数
无注意力机制的重写	0.326
dot 注意力机制的重写	0.349
general 注意力机制的重写	0.388
concat 注意力机制的重写	0.331
tensor 注意力机制的重写	0.361

6. 产品推荐的质量

我们使用 Seq2Seq 模型对用户查询问题进行重写后的语句作为输入，递交给产品推荐系统，并评估产品推荐的质量。首先从测试对中抽取 1000 个查询问题，并使用不同的 Seq2Seq 模型变体生成重写问题。然后，将重写问题提交给产品搜索引擎。同时，我们还使用原始问题查询和标准查询来搜索系统，以便建立产品推荐覆盖率和质量的基准，结果列于表 8.4 中。推荐覆盖率定义为多少查询能返回产品。对于推荐质量，我们会为每个问题的重写版本及其检索的产品采样 3000 对（原始问题，推荐产品），并由一组训练有素的人类评委标记每一对查询和推荐之间的相关性。每个标签的范围是{bad，fair，good，excellent}，

1 Oriol Vinyals and Quoc V. Le. A Neural Conversational Model. CoRR, 2015.

我们将标签为{fair 或 good 或 excellent}的对视为正例。请注意，我们只会向评委提交“原始问题”而非重写问题。评委只是将原始问题与返回的产品进行比较，而不知道产品实际上是使用重写问题来检索的。最后，产品推荐的质量可以基于3000 对中有多少正例标记的比例来推算。

在表 8.4 中，我们看到只有 21.0%的原始问题触发了产品推荐，其中 21.1%的推荐质量基本符合要求。但如果我们用相关搜索查询，会看到更高的产品覆盖率，为 84.7%。这是预期的结果，因为我们已经以这种方式过滤测试集。更重要的是，我们看到更高质量的推荐，占 25.3%。这证实了我们收集数据的方法是有效的：使用者点击的相关搜索查询确实与问题相关，因此，与使用原始问题进行搜索相比，使用重写问题返回的推荐具有一定的相关性。

在 Seq2Seq 模型重写中，使用 general 注意力机制可以实现最高的推荐覆盖率和质量。相对于原始查询，覆盖率增加了 3.5 倍，推荐质量增加了 50%。即使与相关搜索查询（ground truth）相比，推荐覆盖率仅下降 10.7%，但推荐质量上升了 12.4%。

表 8.4　推荐质量

模型	推荐覆盖率	推荐质量
原始问题	21.0%	21.1%
相关搜索查询	84.7%	25.3%
无注意力机制的重写	67.4%	26.7%
dot 注意力机制的重写	64.2%	33.0%
general 注意力机制的重写	74.0%	37.7%
concat 注意力机制的重写	64.1%	18.2%
tensor 注意力机制的重写	64.1%	28.8%

7. 讨论

结果表明，使用适当的训练数据将问题重写可以提高推荐覆盖率和质量，这种分阶段的方法可以无缝应用至现有的基础设施中。它不需要对现有信息检索系统进行任何更改，因为重写的查询问题可以代替原始问题或与原始问题查询一起

递交至推荐系统。此外，仅针对“What”和“How”问题只是迈向通用问答系统的第一步。读者可以想象这个应用程序将成为一个大型、全面的系统的一部分，目前该应用程序只关注产品推荐。最后，读者可能会觉得，尽管观察到了显著的改善，但推荐质量仍然不高。而 6.6.6 节使用 Seq2Seq 模型进行相关性评分可以过滤出更好的推荐。

8.6.6 应用程序 2：相关性评分

Seq2Seq 模型估算推荐产品后验分布的能力非常适合用于控制产品推荐的质量。当从信息检索基础结构返回一系列的推荐时，Seq2Seq 模型可以作为质量过滤器，仅向用户显示最相关的推荐。

1. 训练和数据

我们在搜索数据集上对 Seq2Seq 模型进行了训练，该数据集由从商业产品搜索引擎中抽取的点击（查询，产品）对组成。每个产品都由描述相应产品特征的标题代表，而“查询”是在搜索引擎中由用户输入的。如果用户在使用“查询”进行搜索时点击了系统推荐的产品，我们会将其视为正例（查询，产品）样本对。我们从一个月的点击日志中总共采集了 1500 万次点击，最终得到 640 万个不同的用户查询和 510 万个不同的产品。表 8.5 中总结了训练数据集的统计数据。

表 8.5 相关性评分训练数据集的统计数据

	数量（万）	词汇量（万）	平均长度（个字符）	点击数（万）
查询	640	6.8	4.1	150
产品	510	11.4	9.3	150

2. 模型细节

我们使用了 8.3 节的 Seq2Seq 模型，并在“查询”侧选择了词汇量$|\boldsymbol{V}_q|$= 60k，在“产品”侧选择了$|\boldsymbol{V}_d|$= 100k。不属于词汇表 V 中的任何字被分配符号<UNK>。我们在编码器和解码器侧皆选择词嵌入维度$\boldsymbol{d}_e$= 150，并在解码器侧选择隐藏维度$\boldsymbol{d}_h$= 300。虽然我们确实注意到了更深层网络会对性能有所改善，

但在本实验中，只训练单层 LSTM，以公平地进行下面所述的比较。我们对模型训练了 5 轮。

3. 评估

为了进行定量评估，我们在已由评委标记的测试数据集上进行测试。该测试数据集包含大约 966,000（查询，产品）对，其中每对由一组受过训练的人类评委标记查询和产品之间的相关性。每个标签的范围都是{bad，fair，good，excellent}。

使用 AUC（Area Vnder Curve，ROC 曲线下方的面积大小）作为评估度量，将 good、excellent 标签视为正类，其余标签视为负类。这使得测试集由 234,000 个正对和 731,000 个负对组成，表 8.6 中简要总结了测试数据集的统计数据。我们在产品集上使用统一的先验π，因此如【知识点讲解】概率估计所提的公式（2）中那样，$\Pr(\text{product}|\text{query}) \propto \Pr(\text{query}|\text{product})$。最后我们将$\Pr(\text{query}|\text{product})$作为(query,product)的相关性分数，然后根据测试数据集中所有对的标记计算 AUC。

表 8.6　相关性评分测试数据集的统计数据

查询数量（万）	产品数量（万）	对数（万）	正例数量（万）	副例数量（万）
2.3	91.5	96.5	23.4	73.1

下面与现有的相关性评分基线进行比较，包括 CDSSM[1]和 DeepIntent[2]（详细请见第 6 章），这两者都是基于深度学习为基础的评分模型。给定（查询，产品）对，CDSSM 和 DeepIntent 只使用编码器，将查询和广告单独编码为两个向

1 Shen, Y., He, X., Gao, J., Deng, L., & Mesnil, G. Learning semantic representations using convolutional neural networks for web search. Proceedings of the 23rd International Conference on World Wide Web, 2014：373-374.

2 Zhai, S., Chang, K.-h., Zhang, R., & Zhang, Z. M. (2016). Deepintent: Learning attentions for online advertising with recurrent neural networks. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016：1295-1304.

量，然后直接以这两个向量计算余弦相似度作为相关性得分。我们在相同的数据集上分别用 CDSSM、DeepIntent 和 Seq2Seq 模型来训练，并通过比较同一测试数据集上的 AUC 来评估性能，结果如表 8.7 所示。

表 8.7　不同模型的 AUC 性能

模型	解码器	编码器结构	编码器词嵌入	AUC
(a)CDSSM	None	Conv / max pooling	三字母的单词哈希向量	0.726
(b)DeepIntent	None	Conv / max pooling	单词	0.728
(c)DeepIntent	None	BLSTM/ last pooling	单词	0.798
(d)Seq2Seq	Yes	BLSTM / last pooling	单词	0.840

请注意，CDSSM 模型和 DeepIntent 模型仅使用编码器打分，与 Seq2Seq 模型不同，它既有编码器，也有解码器组件。因此，为了进行公平的比较，有必要保持编码器设置尽可能相似，例如编码器架构、编码矢量大小和递归神经网络的深度。编码矢量大小很容易保持公平，我们在模型中将其设置为 300。所有模型只训练一层的 LSTM 或卷积层（convolution），并将词嵌入向量大小设置为 150（如果适用）。下面介绍不同的编码器架构及其 AUC 性能。请注意，我们没有在 Seq2Seq 模型中使用注意力机制来保持比较的公平。

（1）以 BLSTM 层为基础的 DeepIntent 模型和 Seq2Seq 模型最相似，即它们具有相同的编码器架构。它们基于词嵌入层开始，利用 BLSTM 计算隐藏向量序列，并将最后一个向量作为最终编码向量。因此，将此与 Seq2Seq 进行比较可以公平地证明使用解码器所带来的增益。在表 8.7 中，我们看到 Seq2Seq 模型的 AUC 达到 0.840，如表 8.7 行（d）所示。表 8.7 行（c）中显示 AUC 为 0.798。

（2）CDSSM 模型使用卷积层，而不是在编码器侧使用 BLSTM 层。卷积层通过滑动窗口，每三个词的聚合是基于三个字母的单词哈希向量。卷积层输出的是一系列的向量，透过最大池化将它们进一步缩减为最终编码向量。在使用 CDSSM 模型实现时，它还有一个完全连接的层，以减少最终编码向量的大小，满足在线性能。为了进行公平比较，我们将内部隐藏矢量大小和最终编码矢量大小都设置为 300。我们看到 CDSSM 模型在表 6-7 行（a）中得到了更差的

AUC——0.726。

（3）由于 CDSSM 模型在编码器方面如此不同，即基于三字母的嵌入和卷积层，我们修改了 DeepIntent 模型，改用卷积层，以便了解 AUC 中的丢失来自何处。具体来说，我们想知道它是来自不同的嵌入层还是递归（recurrent）层。在使用具有卷积层的 DeepIntent 模型之后，实现的 AUC 仅为 0.728，如表 6.7 行（b）所示，类似于表 6.7 行（a）中的 CDSSM 模型。比较表 6.7 行（b）与表 6.7 行（c），结果显示基于 BLSTM 层的编码器优于基于卷积层的编码器。

总之，Seq2Seq 模型不仅提供允许概率解释的分数，而且比 CDSSM 和 DeepIntent 等类似的评分模型具有更好的性能。

【知识点讲解】CDSSM 的三个字母的单词哈希向量与卷积层：CDSSM 将每个英文字以三个字母的单词哈希向量表示。例如，boy 可以用拆解成{#bo, boy, oy#}，并用一个约 3 万大小的{0,1}向量表示。哈希向量每个位置代表一个不同三字母的{0,1}出现频率，如图 8.4 所示。这种哈希表现方式，可以避免传统词嵌入的 out-of-vocabulary 问题。英文字的变化很多 （远大于 3 万），但是 3 万的向量足以表示每个英文字。

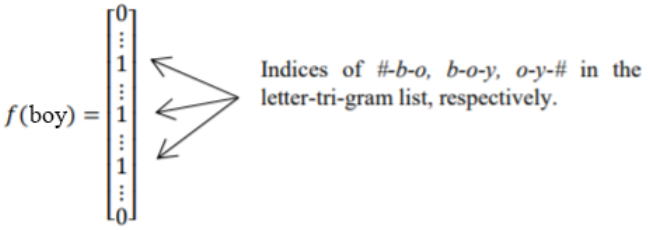


图 8.4　三个字母的单词哈希向量

将每个字用三字母哈希向量表示后，CDSSM 使用卷积层通过滑动窗口，将每三个字母的哈希向量聚合成一个隐藏向量。卷积层输出的是一系列的向量，透过最大池化将它们进一步缩减为最终编码向量。

4. 讨论

下面我们简要讨论一下计算和实现的成本，并指出为什么 Seq2Seq 模型能够成为一个良好的质量过滤器。首先，从实际应用角度来看，让 Seq2Seq 模型在现有信息检索系统之上起作用不需要对基础设施进行修改。然而，Seq2Seq 模型的投影层是比较费时的，当词汇量大时计算量也特别大。但当用作信息检索引擎的出口控制时，返回的产品推荐数量是有限的，通常在几十条的范围内，所以还可以接受。此外，批处理和分层（hierarchical）softmax（类似 SVD 矩阵分解）可以进一步减少所需的计算时间。

8.6.7 应用程序 3：聊天机器人

我们介绍的最后一个应用程序是专门针对产品推荐的聊天机器人，这个应用程序构建在前面所介绍的问题重写与推荐评分的应用之上。虚拟代理和聊天机器人因其用户友好性和互动性而受到欢迎，它们不仅减少了搜索引擎的一些工作，还创建了新的用户交互入口点。[1]下面我们解释一下互动的流程。

1. 对话流和系统行为

交互式会话以用户提交的第一个查询问题开始。例如，用户询问聊天机器人“How do I connect my tablet to TV”，然后聊天机器人从信息检索后端系统中检索相关产品推荐的初始列表。为此，它将 Seq2Seq 模型应用于对用户查询问题的理解并将其进行重写，以转换为标准查询，在本例中重写为“tablet tv connector”。再将该标准查询提交给信息检索系统，该系统返回产品推荐列表，例如，有关 HDMI 线缆、Micro HDMI 线缆或 VGA 线缆的产品。

聊天机器人使用 Seq2Seq 模型的后验分布估计来计算返回的广告列表的分布，并估计其对应的条件熵。在决策步骤中，如果条件熵小于某个阈值 T，则将

1 Jacob Aron. How innovative is Apple’s new voice assistant, Siri? New Scientist 212, 2836 (2011), 24.

最高k（默认为 3）个最相关的广告返回给用户。每个广告包括一张产品图片、销售价格、销售商家，并嵌入了超链接，以便用户点击。用户点击后系统会将用户转向到商家托管的电子商务网页，以便用户可以继续浏览并进行购买。否则，机器人会挑一个有条件的互信息最大化的问题来询问用户。在这种情况下，有一种问题是关于连接器产品的“size”。例如，“what size do you want”，此对话将持续到最终建议为止。图 8.5 以时序图的形式给出了该过程的图示。接下来我们解释如何制定这样的问题。

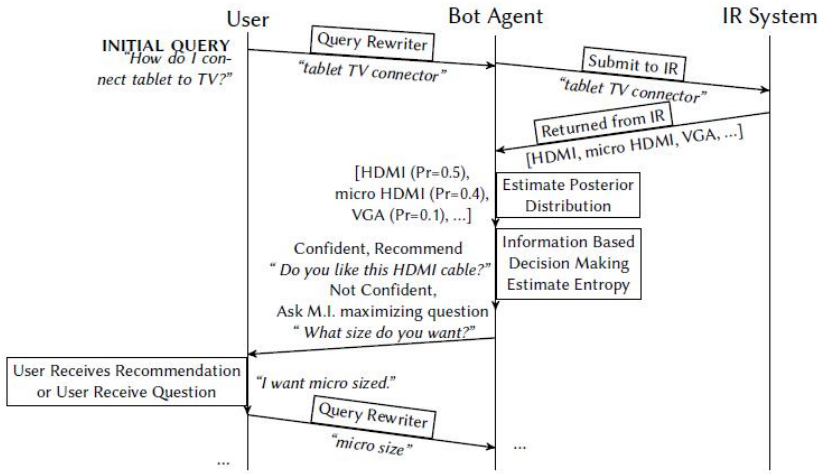


图 8.5　聊天机器人互动设计

2. 问题制定

根据最大化预期信息增益的原则，在每个步骤中，如果聊天机器人不自信，则应该提出有条件的互信息最大化的问题。这里的问题是，最大化的问题是什么？如果我们允许用户提出任意问题，那么聊天机器人可能会遇到以下问题：①问题可能与产品无关，因此令人困惑；②互信息 (mutual information) 难以估计。

为解决此问题，我们会利用与每个产品相关联的属性制定问题。例如，关于笔记本电脑的产品具有“处理器”“RAM 大小”“制造商”等属性。类似地，对于衣服，存在诸如“颜色”“尺寸”和“材料”的属性。根据属性制定问题，上述问题就能解决。

首先，用户更容易理解并合作。用户会认为聊天机器人正在与他们合作，通过确认属性信息来缩小最相关的产品范围。其次，估计互信息和基于属性的问题是直截了当的。请注意，属性仅取决于产品，因此“输入—广告—属性”形成马尔可夫链。这允许我们估计互信息，因为条件分布$\mathbf{Pr(Ad, Attribute|Input}_1^n)$可以通过计算得到：

$$\mathbf{Pr(Ad, Attribute|Input}_1^n) = \mathbf{Pr(Attribute|Ad)}\ \mathbf{Pr(Ad|Input}_1^n)$$

其中第一个因子是通过计数估计的，第二个因子是由 Seq2Seq 模型的后验更新直接提供的。在识别出互信息最大化属性之后，将提出问题并且收集用户输入以再次更新后验分布。例如，如果用户正在寻找笔记本电脑，那么问题可能会是：

你喜欢哪个制造商？

3. 实施和定性反馈

我们使用 Microsoft Bot Framework[1]，这是一个聊天机器人开发工具。它支持各种平台上的机器人对话，包括短信、Skype、Slack、Messenger 等。图 8.6 是以 Skype 为平台的聊天机器人的屏幕截图。

1 Microso. Bot Framework Documentation, Accessed: 2016-12-25.

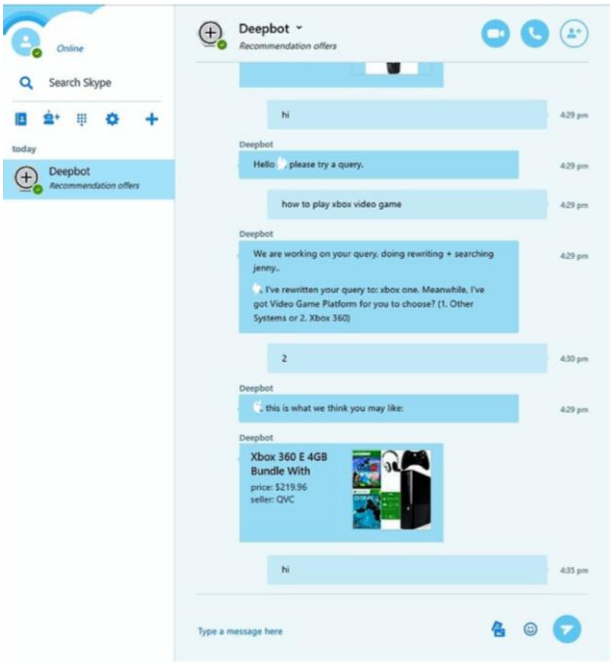


图 8.6　聊天机器人的示例

我曾经向几位同事展示原型，得到了一些令人鼓舞的反馈。他们中的大多数人都对聊天机器人在提出相关问题时推荐产品的能力感到惊讶。“how to connect tablet to tv”的案例也是一个令人振奋的成功经验。当 HDMI 产品被建议给用户时，用户点击链接，随即弹出网页的标题：“16.4ft Ultra-thin Micro HDMI D to A Long　Cable - Connect Tablet / Smart Phone / Mobile / Laptop / Camera to HD TV”。有兴趣的用户可以借此机会了解有关 Micro HDMI 线缆的更多信息（它不仅可以连接平板电脑，还可以将其他设备连接到电视机）并购买。尽管如此，一些同事指出，这个聊天机器人不应该是独立的。除了推荐产品，还应该与其他服务集成，并提供视频教程。最后，一位同事询问聊天机器人是否可以提供除产品以外的其他垂直行业的信息。通过解释系统的工作原理，同事理解了通过不同垂直方向的数据训练，并结合相应的搜索引擎，可以将聊天机器人应用到所需的垂直领域。

8.7 本章小结

在本章我们介绍了 Seq2Seq 模型，这是一个基于序列到序列的模型，用于查询理解、产品推荐和用户交互。通过模型对用户查询的问题进行重写之后，显著提高了推荐覆盖率和质量。对于相关性评分，AUC 是一个关键指标，超过了现有系统。它还展示了更有效的用户交互和聊天机器人设计的巨大潜力，我们可以根据最大化信息增益的原则为用户严格制定问题。作为一项正在进行的工作，我们希望继续在聊天机器人上进行工作和实验，可能还有些定量实验。一个可能的实验方向是我们可以测量用户获得他或她需要的信息的效率（即测量交互轮数）。

参考文献

[1] ARON J. How innovative is Apple's new voice assistant, Siri? New Scientist 2011.

[2] COVER T, THOMAS J. Elements of information theory. 2012. John Wiley & Sons.

[3] DUCHI J, HAZAN E, SINGER Y. Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research[J]. 12, (pp. 2011:2121-2159.

[4] HOCH J, STERN A. Maximum entropy reconstruction. eMagRes. 1996.

[5] HOCHREITER S, SCHMIDHUBER J. Long short-term memory. Neural computation,[J] 9, 1997: 1735-1780.

[6] KANNAN A, KURACH K, RAVI S, et al. Smart Reply: Automated Response Suggestion for Email. Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining 2016 [C].

[7] KHUDANPUR S, WU J. Maximum entropy language model integrating N-grams and topic dependencies for conversational speech recognition. Acoustics, Speech, and Signal Processing, IEEE International Conference 1999[C], 1, 553-556.

[8] LUONG M, PHAM H, MANNING C. Effective Approaches to Attention-based Neural Machine Translation. Proceedings of the Conference on Empirical Methods in Natural Language Processing 2015[C],1412-1421.

[9] Microsoft Bot Framework Documentation. 2016.
[10] PAPINENI K, ROUKOS S, WARD T, et al.. BLEU: A Method for Automatic Evaluation of Machine Translation. Proceedings of the 40th Annual Meeting on Association for Computational Linguistics. Stroudsburg: Association for Computational Linguistics 2012,311-318.
[11] PETERS J. Speech recognition system, training arrangement and method of calculating iteration values for free parameters of a maximum-entropy speech model. Google Patents, 2016.
[12] SHEN Y, HE X, GAO J, et al. Learning semantic representations using convolutional neural networks for web search. Proceedings of the 23rd International Conference on World Wide Web [C] ,2014:373-374.
[13] SKILLING J, BRYAN K. Maximum entropy image reconstruction: general algorithm. Monthly notices of the royal astronomical society [J], 211,1984:111-124.
[14] SOCHER R, CHEN D, MANNING C, et al. Reasoning With Neural Tensor Networks for Knowledge Base Completion. Advances in Neural Information Processing Systems 2013 [C], 26 ,926-934.
[15] SOCHER R, PERELYGIN A, WU J, et al. Recursive deep models for semantic compositionality over a sentiment treebank. Proceedings of the conference on empirical methods in natural language processing (EMNLP) 2013 [C], 1631 – 1642.
[16] SUTSKEVER I, VINYALS O, LE Q. Sequence to sequence learning with neural networks. Advances in neural information processing systems [C] ,2014:3104-3112.
[17] Team Development Theano. Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints 2016, abs/1605.02688.
[18] VINYALS O, LE Q. A Neural Conversational Model. CoRR 2015, abs/1506.05869.
[19] ZEILER M. ADADELTA: an adaptive learning rate method. arXiv preprint 2012. arXiv:1212.5701.
[20] ZHAI S, CHANG K, ZHANG R, et al. Deepintent: Learning attentions for online advertising with recurrent neural networks. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 2015[C], 1295-1304.

Deep Learning Model and Application

第 9 章

word2vec 的改进：fastText 模型

fastText 模型是 Facebook AI Research（FAIR）实验室在 2016 年开源的一个词向量生成及基于词嵌入向量模型（word embedding model,简称词向量）的文本分类工具。fastText 的模型架构与 word2vec 基本一致，可以认为是 word2vec 的一种扩展模型，其核心思想是运用 subword 信息重建 word2vec 的字典，从而可以在实质上扩大 word2vec 字典的表达能力，并使得稀有词（rare word）和未知词（unknown word）的词向量也可以比较好地被计算出来。此外，顾名思义，fastText 模型的一大特点就是快，它可以在超大规模数据集上进行快速训练，非常便于在各种应用场景中使用。关于 word2vec 的基本原理，读者可以参考本书第 3 章的内容。

9.1 fastText 模型的原理

fastText 模型的原理与 word2vec 非常相似。根据第 3 章的内容我们知道 word2vec 模型有 CBOW 和 Skip-Gram 两种算法，本章我们以 Skip-Gram 算法为例介绍 fastText 模型。

9.1.1 回顾 Skip-Gram 算法

假设一个文本数据集的字典大小是W，也就是说这个文本数据集里面的所有单词都来自字典里的这W个单词。我们用正整数$w \in \{1,2,\ldots,W\}$来表示W个单词的索引。我们的目标是要将字典里的每个单词w学习出一个词向量来。假定文本数据集由一个单词序列组成，即$w_1, w_2, \ldots, w_T$，Skip-Gram 算法的目标是最大化下面的这个对数似然函数（log-likelihood）：

$$\sum_{t=1}^{T}\sum_{c\in C_t}\log p(w_c|w_t)$$

其中，C_t包含了w_t的上下文单词的索引。也就是说，我们希望通过当前单词w_t去预测它的上下文单词w_c。假设有一个可以将单词对(w_t, w_c)映射成实数值分数

的打分函数s，那么就可以用下面的归一化指数函数（ softmax 函数）形式的公式来定义$p(w_c|w_t)$，即给定w_t后，w_c出现在w_t上下文的概率：

$$p(w_c|w_t)=\frac{\mathrm{e}^{s(w_t,w_c)}}{\sum_{j=1}^{W}\mathrm{e}^{s(w_t,j)}}$$

由于w_t的上下文单词一般有多个，所以我们可以把预测每个上下文单词的任务作为一个独立的二分类问题。具体对于w_t来说，则是把它的上下文单词w_c作为二分类问题的正例，把字典中所有没有出现在w_t的上下文的单词作为负例。由于字典一般非常大，我们可以从中随机抽取一个比较小的集合$\boldsymbol{N}_{t,c}$来当作负例集合。参照二元逻辑回归的损失函数（loss of function binary logistic regression），可以得到下面的负对数似然函数：

$$\log\left(1+\mathrm{e}^{-s(w_t,w_c)}\right)+\sum_{n\in N_{t,c}}\log\left(1+\mathrm{e}^{s(w_t,n)}\right)$$

假如我们把二元逻辑回归的损失函数记为$l:x\rightarrow\log(1+\mathrm{e}^{-x})$，那么可以把 skip-gram 模型的目标函数重写为：

$$\sum_{t=1}^{T}\left[\sum_{c\in C_t}l\left(s(w_t,w_c)\right)+\sum_{n\in N_{t,c}}l\left(-s(w_t,n)\right)\right]$$

接下来的问题就是如何计算打分函数$s(w_t,w_c)$，一个很自然的想法就是用词向量来计算。对于单词w，我们定义它的两个维度为d的词向量$\boldsymbol{u}_w$和$\boldsymbol{v}_w$，分别对应于第 3 章中的输入词向量（input vector）和输出词向量（output vector）。那么，我们就用$\boldsymbol{w}_t$的输入词向量和$\boldsymbol{w}_c$的输出词向量的标量积来定义打分函数，即$s(w_t,w_c)=\boldsymbol{u}_{w_t}^{\mathrm{T}}\boldsymbol{v}_{w_c}$。这就是我们在第 3 章里介绍过的 Skip-Gram 算法。

9.1.2 subword 模型

我们注意到，在 word2vec 模型里（包含 CBOW 和 Skip-Gram），字典中的每一个单词都会用一个单独的词向量来表示。

这样做会带来一些问题。

首先，word2vec 本质上还是一种统计机器学习的方法，因而对于在训练集中的高频词（frequent word）来说，它们最终训练出来的词向量的质量会比较好。相反，对于出现次数比较低的稀有词（rare word）来说，它们最终训练出来的词向量的质量会比较差。我们知道，在文本数据集中，词频一般遵从幂律分布（power law distribution），稀有词的平均词频很低但是总数不低，因而它们的词向量质量对于整个 word2vec 模型输出的词向量全集来说依然是很重要的。

其次，word2vec 只对字典中出现的词计算词向量，对于未知词没有提供合理的计算方法，这对很多应用场景来说非常不够。比如在搜索广告应用中，新词每时每刻都在产生，它可能是某个新公司的名称，可能是某个新产品的商标，也可能是某种产品的新型号。这些新词对于 word2vec 模型来说都是未知词，如果我们没有针对这些未知词在建模时进行相应的处理，就无法得到它们的高质量词向量。

对于稀有词，我们还可以通过增加其采样比率来提升训练效果。对于未知词，我们应该采取什么样的策略呢？不妨思考一下，当一个人遇到不认识的单词时，他是怎样来猜这个词的含义的。从语义认知学的角度，主要有 3 种方法。

第 1 种称为重新编码（recoding），其核心方法是根据一个单词不同组成部分的含义来重新构建出整个单词的意思。比如单词 psychology（心理学），其中 psy 有 know（知道）和 study（研究）的含义，cho 有 mind（精神）和 soul（心灵）的含义，logy 是表示学科领域的词尾。那么如果我们把这 3 个部分的含义组合起来就得到“研究心灵的学科”这样一个比较笼统的解释，这也就和“心理学”的含义非常接近了。这种方法比较适用于一些合成词，它们的每个组成部分的含义比较明确，从而易于重构解读，类似的词还有 homework、housewife、handyman 等。

第 2 种称为类推（analogizing），其核心方法是找到某种构词的规律，然后

推理出单词的含义。比如单词 admob，这是一个公司的名字，我们根据构词规律可以把它拆成 ad 和 mob 两个部分，而这两部分根据常识我们知道它们恰好是 advertising 和 mobile 的简写，因而就可以比较从容地猜出这个叫 admob 的公司的主营业务与移动端广告有关。这种方法比较适用于一些新出现的品牌、商标、产品名，以及新造的词汇等。

第 3 种称为预测（prediction），其核心方法是根据单词的上下文来预测未知词的含义。比如下面这句话：

Bougainvillea was voted as the city flower last sunday.

其中，bougainvillea（三角梅花）这个单词我们不认识，但是通过对整句话的理解，可以很容易地预测出这应该是某种比较受欢迎的花。很多时候，我们理解到这里就足够了，至于它到底是什么花可能并不重要。这种方法使用的范围最为广泛，在人们阅读时经常不自觉地使用。

由此我们可以看出，word2vec 其实是运用了预测的方法，它利用滑动窗口里的中心词和上下文单词来互相预测含义。那么，我们能否把重新编码和类推的方法也考虑进来呢？答案是肯定的，fastText 算法里应用了一种称为 subword 模型的策略来引入这两种方法。

subword 的意思是次级词汇或者子词（指的是词的某一部分）。例如，goodwill 里 good 和 will 都可以被看作是 goodwill 的 subword。当然，subword 的形式可以有多种选择，像上面这种把合成词拆开的形式只是其中之一，当然大部分词并非合成词，所以无法拆成若干个已经存在的词。fastText 算法里使用的 subword 模型是基于字符n-gram 的（character n-gram）。具体来说，对于一个单词，首先在其首尾各加一个特殊字符“#”用来表示词的开始和结束，然后找出其中所有的n-gram 作为 subword 加入字典里，再把单词本身也当作 subword 加进字典里，这个单词最终会由当作 subword 的词向量来一起表示。举例来说，如单词 heart 和$n = 3$的情况，首先把 heart 变成#heart#，然后找出其中所有的字

符 3-gram，共有如下 5 个：

#he, hea, ear, art, rt#

再加上#heart#一共是 6 个 subword，heart 的词向量就由这 6 个 subword 的词向量来共同表示。在实践中，fastText 的 subword 模型使用了单词里包含的所有字符 3-gram 到字符 6-gram。除此之外，我们还有许多其他的选择，比如可以使用英文单词的前缀（例如 pre、sub、re 等）和后缀（例如 ful、ly、tion、ing 等）来作为 subword。更多的 subword 形态的选择可以参考 9.1.3 节。

在准备好了 subword 的字典以后，接下来的一步就是要考虑如何用 subword 的字典来计算打分函数$s(w_t, w_c)$。假定一个单词w_t的所有 subword 组成的集合记为$\boldsymbol{G}_{w_t}$，其中每个 subword 的$g \in \boldsymbol{G}_{w_t}$的词向量记为$\boldsymbol{z}_g$，这样就可以用这些 subword 的词向量来表示单词w_t了。比如用字典来计算打分函数$s(w_t, w_c)$可以表示为下面的形式：

$$s(w_t, w_c) = \sum_{g \in \boldsymbol{G}_{w_t}} \boldsymbol{z}_g^{\mathrm{T}} \boldsymbol{v}_{w_c}$$

这样一来，我们就可以在单词之间共享 subword 的信息，使训练出来的 subword 的词向量更加准确。对于稀有词和未知词，虽然它们本身没有太多的训练样本，但仍然可以用它们的 subword 的词向量来计算这些词的词向量。具体来说，稀有词的 subword 可以和其他词的 subword 共享，从而在客观上间接地增加稀有词的训练样本；未知词的词向量可以通过叠加它的所有 subword 的词向量来得到。这样就解决了稀有词和未知词的词向量难以计算的问题。

9.1.3 subword 形态

subword 形态的选择非常灵活，在 9.1.2 节中我们提到可以使用单词本身、字符n-gram、单词前缀/后缀等形态。

除此之外，还可以考虑根据单词形态学（morphology）来得到更丰富的信

息，比如一个非常复杂的稀有词 antidisestablishmentarianism（反政教分离运动），可以根据单词形态学中的形态树（morphology tree）切分成 anti-dis-establish-ment-arian-ism 的形式，其中每个构词单元都可以作为这个词的 subword。

另外，还可以从单词拼读的角度，根据音节（syllable）的划分来得到 subword。比如，根据单词音标的拼读，car、 hotel、beautiful、oxymoron、antidisestablishmentarianism 的音节划分分别是：car、ho-tel、beau-ti-ful、ox-y- mor-on、an-ti-dis-es- tab-lish-ment- ar-i-an-is-m。

无论是形态学划分还是音节划分，我们都可以使用一些现成的工具包来完成。

对于英文、法文、德文等表音文字来说，上述的 subword 形态基本够用了。但是对于像中文这种表意文字，就需要使用一些其他的 subword 形态了。一种比较简单的方式是首先把中文语句进行分词，然后把每个词作为单词（word），再把单词中的每个单字作为 subword。比如，“高考成绩查询”经过分词以后得到 3 个单词：“高考”“成绩”“查询”，然后单词中的每个单字就作为这个单词的 subword。除此之外，也有研究人员从汉字构成的角度探索使用偏旁部首作为 subword 的可能性。

9.1.4 分层 softmax

我们知道，word2vec 的输出层需要做一个 softmax 回归，而 fastText 模型把它改进成为分层（hierarchical）softmax，从而大大提升了运算速度。

首先，回顾一下逻辑回归（logistic regression）的知识。假设在一个二分类问题中，有m个标注样本$\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$，其中$\boldsymbol{x}_i$是样本的特征表达向量，$y_i \in \{0,1\}$。假定分类函数有如下形式，其中$\boldsymbol{\theta}$为参数向量：

$$h(\boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{1 + \mathrm{e}^{-\boldsymbol{\theta}^{\mathrm{T}}\boldsymbol{x}}}$$

那么逻辑回归的损失函数一般可以写为：

$$L(\boldsymbol{\theta})=\sum_{i=1}^{m}\left(y_i\log h(\boldsymbol{x}_i,\boldsymbol{\theta})+(1-y_i)\log\left(1-h(\boldsymbol{x}_i,\boldsymbol{\theta})\right)\right).$$

在 softmax 中，类别数是大于 2 的，即$y_i\in\{0,1,\ldots,K\}$。对于输入样本$\boldsymbol{x}_i$，我们需要输出一个K维向量，其中每个元素表示样本$\boldsymbol{x}_i$输入相应类别的概率。这样，分类函数和损失函数分别写为：

$$h(\boldsymbol{x},\boldsymbol{\theta})=\begin{bmatrix}p(y=1|\boldsymbol{x};\boldsymbol{\theta})\\p(y=2|\boldsymbol{x};\boldsymbol{\theta})\\\vdots\\p(y=K|\boldsymbol{x};\boldsymbol{\theta})\end{bmatrix}=\frac{1}{\sum_{j=1}^{K}\mathrm{e}^{\boldsymbol{\theta}^{(j)\mathrm{T}}\boldsymbol{x}}}\begin{bmatrix}\mathrm{e}^{\boldsymbol{\theta}^{(1)\mathrm{T}}\boldsymbol{x}}\\\mathrm{e}^{\boldsymbol{\theta}^{(2)\mathrm{T}}\boldsymbol{x}}\\\vdots\\\mathrm{e}^{\boldsymbol{\theta}^{(K)\mathrm{T}}\boldsymbol{x}}\end{bmatrix}.$$

$$L(\boldsymbol{\theta})=\sum_{i=1}^{m}\sum_{k=1}^{K}\mathrm{sgn}(y_i=k)\log\frac{\mathrm{e}^{\boldsymbol{\theta}^{(k)\mathrm{T}}\boldsymbol{x}_i}}{\sum_{j=1}^{K}\mathrm{e}^{\boldsymbol{\theta}^{(j)\mathrm{T}}\boldsymbol{x}_i}}.$$

其中，$\mathrm{sgn}(\cdot)$为示性函数。可以看出，在K比较大时，softmax 的运算量非常大，因为对每一个样本，它都要计算K个概率$p(y=j|\boldsymbol{x};\boldsymbol{\theta}),j=1,\ldots,K$，并做归一化处理。为了解决这一问题，我们就需要引入分层 softmax。其基本思想就是使用树结构来替代标准 softmax 的平层结构，使得在计算$p(y=j|\boldsymbol{x};\boldsymbol{\theta})$时，我们只需要计算树上某一条路径里所有的节点的概率值就可以了。

分层 softmax 一般选用根据类别标识的频次构造的霍夫曼树，如图 9.1 所示。其K个叶子节点表示每个类别，$K-1$个内部节点作为参数。为了计算$p(y=j|\boldsymbol{x};\boldsymbol{\theta})$，只需要计算从根节点到叶子节点$y_j$ 的路径。假定路径长度为$s(y_j)$，那么有：

$$p(y=j|\boldsymbol{x};\boldsymbol{\theta})=\prod_{l}^{s(y_j)-1}\sigma\left(\zeta\left(n(y_j,l+1),LC\left(n(y_j,l)\right)\right)\boldsymbol{\theta}_{n(y_j,l)}^{\mathrm{T}}\boldsymbol{x}\right)$$

其中，$n(y_j,l)$表示路径$s(y_j)$上的某个中间节点，$LC\left(n(y_j,l)\right)$表示$n(y_j,l)$的左子节点，$\sigma(\cdot)$是 sigmoid 函数：

$$\zeta(a,b)=\begin{cases}1 & a=b\\ -1 & a\neq b\end{cases}$$

在图 9.1 所示的例子中，从根节点到y_2的路径实际上进行了三次二分类的逻辑回归。因而，通过分层 softmax，可以把对$p(y=j|\boldsymbol{x};\boldsymbol{\theta})$的计算复杂度从$K$降到$\log K$。

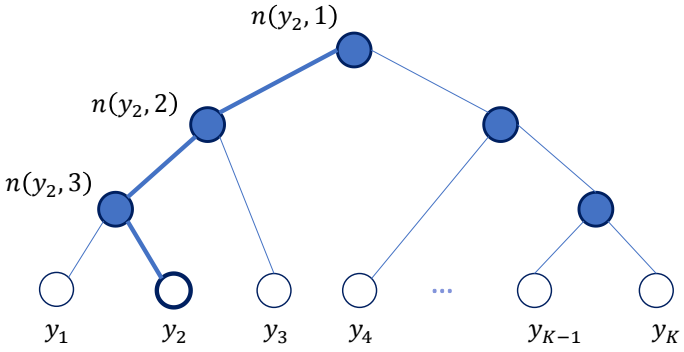


图 9.1　分层 softmax 示例

9.1.5　fastText 的模型架构

fastText 模型有两种用法，一种用来训练词向量，另一种用来做文本分类。用来训练词向量的 fastText 模型架构与 word2vec 几乎完全一样，唯一不同的地方就是打分函数$s(w_t,w_c)$的计算方式，这在 9.1.4 节我们已经做了重点介绍，这里就不赘述了。用来做文本分类的 fastText 模型架构如图 9.2 所示，这个结构和 word2vec 的 CBOW 模型非常相似，都是只有输入层、隐藏层、输出层的三层简单结构。下面我们以情感分析为例来介绍这个模型结构。

所谓情感分析，就是对一段文本所表达的情感进行分类，比如在互联网购物平台上一般都允许顾客对平台售卖的商品进行评论，平台可以根据大量顾客的评论来对其售卖的商品或者相关的服务进行优化和调整。这里面就涉及对商品评论的情感分析，简单来说，就是需要对每一条评论（可以认为是一句话）所表达的情绪进行分类，比如正面（“这款鼠标好看又好用，物美价廉，赞！”）、负面（“买

回家用了第二天就坏了，质量真差!”）或者中立（“我是来看评论的”）。

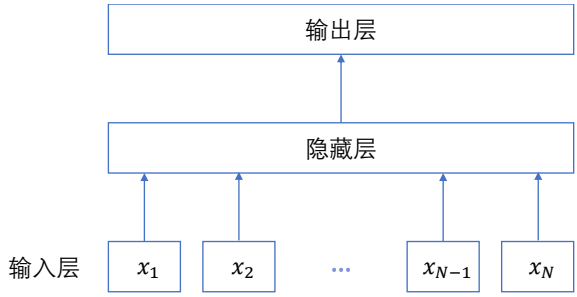


图 9.2　fastText 的模型架构

在 fastText 文本分类模型中，我们把一句话中所有单词的 subword 抽取出来，然后把这些 subword 的词向量作为输入层的输入特征，经过隐藏层之后，在输出层用分层 softmax 来预测分类标签（正面、负面、中立）。分层 softmax 可以大大降低模型训练的时间，这在 9.1.4 节我们有所介绍。从图 9.1 我们可以看到，fastText 文本分类模型的核心思想是把输入文本中的词及其 subword 叠加得到文本的表达向量，然后对这个表达向量进行 softmax 多分类。

9.1.6　fastText 算法实现

fastText 的算法实现与 word2vec 非常相似，大致框架可以参考第 3 章的内容。主要区别有两点：一是选用某种 subword 模型来建立 fastText 的词典；二是打分函数$s(w_t, w_c)$的计算。这些内容在 9.1 节里已经详细解释，不再赘述。

fastText 的优点是可以在大规模数据集合上进行快速训练，比如，在使用标准多核 CPU 的情况下，10 分钟内即可处理超过 10 亿个词汇，因此，它在很多应用场景下被广泛使用。所以，继 word2vec 之后，faxtText 已经成为在文本挖掘和自然语言处理领域里作为词向量或者进行词向量初始化的标准模型。此外，Facebook 还公开了由 fastText 预训练好的 100 多种语言的词向量供用户使用，更是为 fastText 模型的推广铺平了道路。

9.2 应用场景：搜索广告中的查询词关键词匹配问题

在搜索广告中，用户的查询词（query）需要与广告主的关键词（keyword）进行匹配，从而在广告数据库中抽取出相关的广告进行点击率预测和排序。除一部分关键词可以通过精确匹配（exact match）来与查询词进行关联外（例如查询词和关键词都是 iPhone XS），还有大量的关键词与查询词在字面上并不一致，但是在语义上或者用户查询意图上有很强的关联性（例如查询词和关键词分别是 smart phone 和 iPhone XS）。对于这部分关键词的匹配，我们就需要用到智能匹配（smart match）算法。

智能匹配算法的应用主要有 3 种方法。

第 1 种方法是通过对搜索广告系统的日志进行分析，从中挖掘出一些相关联的查询词和关键词；

第 2 种方法是把查询词和关键词分别作为用户和广告主的“语言”，训练一个机器翻译模型，从而可以把查询词“翻译”成关键词；

第 3 种方法是把查询词和关键词都映射到一个线性空间里，然后找出距离查询词比较近的关键词。

对于上述第 3 种方法，我们又有多种选择。

首先，可以考虑用向量空间模型（vector space model）来表示查询词和关键词。这种方法的优点是简便而直观。缺点有两个：一是向量空间维度很大（词典大小）；二是很难匹配语义接近但词汇不同的查询词和关键词。

其次，可以使用一些词嵌入向量（word embedding，word vector）模型将单词从高维空间里的稀疏表示向量映射成为低维空间里的稠密表示向量。主成分分析（principal component analysis）可以达到上述目的，但是主成分分析需要计算一个超大规模矩阵的奇异值分解（singular value decomposition），所以不太适合这个应用场景。于是，我们考虑用 word2vec 模型来训练词向量，避免进行超大规模矩阵的分解运算。再考虑到搜索广告中需要对大量稀有词和未知词的

词向量进行计算，我们最终选择 fastText 模型来训练查询词和关键词的词向量，然后通过计算查询词和关键词在词向量空间里的欧氏距离来对它们进行匹配。

在这个应用场景里，我们遇到的主要问题是训练数据的构建。在 fastText 模型中，训练数据是文本流。假定有一个滑动窗口在文本流上从前向后滑动，在窗口里面的某个单词w_t和它的所有上下文单词w_c就组成了一个训练样本。在搜索广告的日志中，我们可以抽取一段时间内用户点击广告的数据。具体来说，用户提交了一个查询词q并点击了某一个广告，与这个广告相关联的关键词为k，那么我们就抽取出一个点击对(q,k)。在一段时间内，同样一个查询词可能有若干个这样的点击对（包含重复的点击对）$\{(q,k_1),(q,k_2),\dots,(q,k_n)\}$。我们将这些点击对拼在一起构造成一句人造语句$q,k_1,q,k_2,\dots,q,k_n$，再把这段时间内所有的查询词构造的人造语句连接起来形成一个文本流数据。这样，我们就可以让滑动窗口在这个文本流上滑动，从而训练 fastText 模型。在训练出的模型中，我们有每个 subword 的词向量表达。在使用时，我们可以把查询词q中的每一个单词进行 subword 拆分，然后把相应的 subword 的词向量计算平均来作为这个单词的词向量，最后把每个单词的词向量再次加权平均从而得到查询词的词向量。对于关键词k的词向量计算也可以类似进行。这样一来，我们就可以通过计算查询词和所有关键词在词向量空间中的欧氏距离来找到与查询词含义最接近的一批关键词进行匹配了。

9.3 本章小结

本章主要介绍了以下内容。

（1）word2vec 的扩展模型 fastText 模型，运用各种 subword 信息重建 word2vec 的字典，这样可以扩大 word2vec 字典的表达能力，从而使得稀有词和未知词的词向量也可以比较好地被计算出来。

（2）以搜索广告中查询词关键词的匹配问题为例，介绍 fastText 算法的应用。

有些读者可能会有疑问，fastText 在模型的角度只是对 word2vec 的一个改进，无论是算法层面还是理论层面看起来都不是很大的创新，那我们为什么要单独拿出一章来介绍它呢。这是因为 fastText 从实际使用效果和训练效率上来看都是非常优秀的文本表示和文本分类工具，它非常适合用于工业界的大规模文本分类问题，已经成为被业内人士广泛认可的标准算法。

此外，fastText 的优秀表现也引发了我们的一个观察，神经网络似乎并不总是越深越好，因为 fastText 和 word2vec 一样都不是很深的神经网络。也就是说，对于文本分类这种偏线性的数据集，深层神经网络相对于浅层神经网络来说优势似乎并不明显。所以说，我们还是需要具体问题具体分析，即使是同样的一个算法，在不同的应用或者不同大小的数据集上的表现也可能是千差万别的，需要静下心来通过对问题的解读、对数据的分析，以及对模型的调整来找到最有效的算法。

参考文献

[1] MIKOLOV T, SUTSKEVER I, CHEN K, et al, Distributed representations of words and phrases and their compositionality, Advances in neural information processing systems 2013[C] ,3111-3119.

[2] MIKOLOV T, CHEN K, CORRADO G, et al, Efficient estimation of word representations in vector space, arXiv preprint arXiv:1301.3781, 2013.

[3] BOJANOWSKI P, GRAVE E, JOULIN A, et al, Enriching word vectors with subword information, Transactions of the Association for Computational Linguistics 2017[C] ,5(2017), 135-146.

[4] JOULIN A, GRAVE E, BOJANOWSKI P, et al, Bag of tricks for efficient text classification, arXiv preprint arXiv:1607.01759, 2016.

Deep Learning Model and Application

第 10 章
生成对抗网络

生成对抗网络（Generative Adversarial Network，GAN）是 Goodfellow 在 2014 年提出来的一种基于博弈论中二人零和游戏的机器学习的新框架。近些年来研究者们对它的关注日益增加，每年都有大量的研究成果涌现出来。

10.1 生成对抗网络的原理

生成对抗网络的基本架构是根据博弈论中的二人零和游戏来设计的。在二人零和游戏中，游戏双方的利益总和是一个常数，即当一方的利益增加时，另一方的利益会相应减少。在生成对抗网络中，游戏的双方分别是一个生成模型（generative model）和一个判别模型（discriminative model）。其中，生成模型负责产生与样本数据尽可能相似的假样本，而判别模型负责辨别一个样本是来自真实样本数据还是来自生成模型产生的假样本数据。生成模型的目标是要通过不断地训练来产生以假乱真的假样本，而判别模型的目标是要通过不断地训练从数据中尽可能把假样本甄别出来。最终，经过多轮调整，生成模型产生的假样本已经让判别模型无法区分，这样就达到了生成对抗网络的目标，即学习到一个可以产生和真实样本数据同分布的数据的生成模型。生成对抗网络中，生成模型和判别模型有多种模型选择方式，一般会选择非线性映射函数模型（可以是各种深度学习模型）来充当生成模型和判别模型。

10.1.1 GAN 的基本模型

在生成对抗网络中，我们要同时训练一个生成模型G和一个判别模型D。生成模型G从一个预先定义好的噪声分布$p_z(\boldsymbol{z})$中随机抽取一个噪声向量$\boldsymbol{z}$作为输入，然后产生一个假样本$G(\boldsymbol{z};\theta_g)$，其中$\theta_g$包含了生成模型的参数。判别模型$D(\boldsymbol{x};\theta_d)$把真实样本和假样本分别作为正例和负例输入，并对它们进行二分类的判别，其中θ_d包含判别模型的参数。生成模型和判别模型的训练是交替进行的：在训练生成模型时，固定判别模型，通过优化参数θ_g来最小化损失函数$\log\big(1-D(G(\boldsymbol{z}))\big)$；

在训练判别模型时，固定生成模型，通过优化参数θ_d来最小化损失函数$\log(1-D(x))$。也就是说，生成模型和判别模型在进行一场极小极大化价值函数$V(G,D)$的二人博弈，即进行如下优化：

$$\min_G \max_D V(G,D) = E_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D(G(\boldsymbol{z}))\right)\right]$$

生成对抗网络的目标是让生成模型可以产生与真实数据几乎没有区别的样本。假定真实数据的分布函数是$p_{\text{data}}(\boldsymbol{x})$，生成模型产生的假样本的分布函数是$p_g(\boldsymbol{x})$，那么我们期望最终得到的生成模型可以使得$p_g(\boldsymbol{x}) = p_{\text{data}}(\boldsymbol{x})$，即假样本可以达到以假乱真的目的。

下面我们解释一下上述极小极大化价值函数的意义。

首先，对于从真实数据分布$p_{\text{data}}(\boldsymbol{x})$中随机抽取的样本$\boldsymbol{x}$，判别模型$D(\boldsymbol{x};\theta_d)$应该判定它为正例，即$D(\boldsymbol{x};\theta_d)=1$，所以应当优化参数$\theta_d$来找到可以极大化$E_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})]$的判别模型$D$，其中$E$表示取数学期望。然后，对于从噪声分布$p_z(z)$中随机抽取的一个噪声向量$\boldsymbol{z}$，生成模型会产生一个假样本$G(\boldsymbol{z};\theta_g)$，对于这个假样本，判别模型$D(G(\boldsymbol{z};\theta_g);\theta_d)$应该判定它为负例，即$D(G(\boldsymbol{z};\theta_g);\theta_d)=0$，所以我们应当优化参数$\theta_d$来找到可以极大化$E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D(G(\boldsymbol{z}))\right)\right]$的判别模型$D$。综合上述两点，我们知道判别模型$D$的目标是要极大化下面的函数，从而达到甄别真实样本和假样本的目的。

$$\max_D V(G,D) = E_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D(G(\boldsymbol{z}))\right)\right]$$

而生成模型G的优化目标与判别模型恰恰相反，它需要达到以假乱真的目的，所以它就要极小化上面的目标，这样就有了一个极小极大化价值函数$V(G,D)$的博弈过程。

$$\min_G \max_D V(G,D) = E_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D(G(\boldsymbol{z}))\right)\right]$$

图 10.1 展示了生成对抗网络的基本结构。

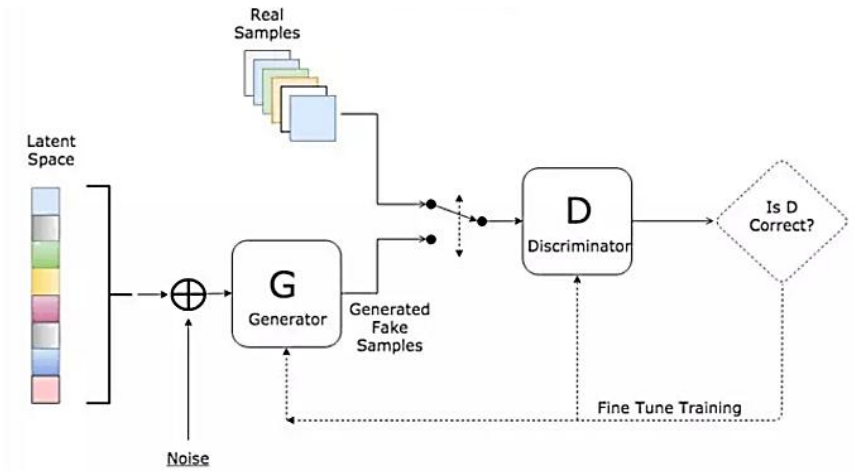


图 10.1　生成对抗网络的基本结构

10.1.2　GAN 优化目标的原理

在 10.1.1 节，我们以二人零和游戏的方式引出了 GAN 的极大极小目标函数；这一节，我们要对这一目标做更进一步的解释，从而便于读者理解其背后的原理。

我们从交叉熵（cross-entropy）开始讨论。在信息论中，常用交叉熵来判别信号分布的相似性。假定p和q是相同事件测度的两个概率分布，其中p为真实样本分布，q为生成样本分布，那么交叉熵的定义如下：

$$H(p,q) = -\sum_i p_i \log q_i$$

可以看出，两个分布p和q越相似，$H(p,q)$就越小。如果 10.1.1 节中介绍的判别模型D是一个二分类器，假定y_1为真实样本分布，那么生成样本的分布就是$(1-y_1)$，再假定$D(x_1)$为判别样本为真实样本的概率，那么判别样本为生成样本的概率就是$\big(1-D(x_1)\big)$。这样，我们可以根据交叉熵的定义写出下式作为优化目标：

$$H\big((x_1,y_1),D\big) = -y_1 \log D(x_1) - (1-y_1)\log\big(1-D(x_1)\big)$$

将上式推广为N个样本后，可得下式：

$$H((x_i,y_i)_{i=1}^N,D)=-\sum_{i=1}^{N}y_i\log D(x_i)-\sum_{i=1}^{N}(1-y_i)\log\big(1-D(x_i)\big)$$

我们知道，在生成对抗网络中，样本点x_i要么来自真实样本分布$p_{\text{data}}(x)$，要么来自生成模型生成的样本分布$G(z)$。对于$x_i\sim p_{\text{data}}(x)$，我们希望判别它为真实样本分布$y_i$；对于$x_i\sim G(z)$，我们希望判别它为生成样本分布$(1-y_i)$。在理想的状况下，生成模型已经可以以假乱真，即达到$y_i=1/2$。将上面的式子写成数学期望的形式并令$y_i=1/2$就可以得到：

$$H((x_i,y_i)_{i=1}^{\infty},D)=-\frac{1}{2}E_{x\sim p_{\text{data}}(x)}[\log D(x)]-\frac{1}{2}E_{z\sim p_z(z)}\left[\log\left(1-D\big(G(z)\big)\right)\right]$$

这样，我们就需要找到一个最优的D来极小化$H((x_i,y_i)_{i=1}^{\infty},D)$，这恰恰等价于极大化下式：

$$\max_D V(G,D)=E_{x\sim p_{\text{data}}(x)}[\log D(x)]+E_{z\sim p_z(z)}\left[\log\left(1-D\big(G(z)\big)\right)\right]$$

而生成模型的目标与判别函数相反，这样就有了下面的极大极小目标函数：

$$\min_G\max_D V(G,D)=E_{x\sim p_{\text{data}}(x)}[\log D(x)]+E_{z\sim p_z(z)}\left[\log\left(1-D\big(G(z)\big)\right)\right]$$

由此可以看出，$V(G,D)$相当于表示真实样本和生成样本的差异程度，$\max_D V(G,D)$就是在固定G的前提下找到能够最大化地判别出样本来源的D，然后在固定D的前提下找G，使得G生成的样本能够最小化与真实样本的差异。

10.1.3 GAN 的训练

在生成对抗网络的训练过程中，对判别模型和生成模型的优化是交替进行的。在每一次迭代中，首先优化判别模型：

（1）从真实数据分布$p_{\text{data}}(x)$中随机抽取m个样本$\{x^1,x^2,\dots,x^m\}$；

（2）从噪声分布$p_z(z)$中随机抽取m个噪声样本；

（3）将m个噪声样本输入生成函数G产生m个假样本$\{\tilde{x}^1, \tilde{x}^2, \ldots, \tilde{x}^m\}$；

（4）在价值函数$V(G,D)$中以均值代替期望，计算近似函数$\tilde{V}(\theta_d) = \frac{1}{m}\sum_{i=1}^{m}\log D(x^i;\theta_d) + \frac{1}{m}\sum_{i=1}^{m}\log\left(1 - D(\tilde{x}^i;\theta_d)\right)$；由于我们要极大化$\tilde{V}(\theta_d)$，需要用下式更新判别函数的参数：$\theta_d \leftarrow \theta_d + \eta\nabla\tilde{V}(\theta_d)$。

然后优化生成模型：

（1）从噪声分布$p_z(z)$中随机抽取另外m个噪声样本$\{z^1, z^2, \ldots, z^m\}$；

（2）在价值函数$V(G,D)$中以均值代替期望，计算近似函数$\tilde{V}(\theta_g) \leftrightarrow \frac{1}{m}\sum_{i=1}^{m}\log\left(1 - D\left(G(z^i;\theta_g)\right)\right)$（价值函数第一项与$\theta_g$无关，可以略去）；由于要极小化$\tilde{V}(\theta_g)$，需要用下式更新生成函数的参数：$\theta_g \leftarrow \theta_g - \eta\nabla\tilde{V}(\theta_g)$。

其中η是学习率。根据经验，在上述过程中，一般对判别模型的优化进行k次，然后对生成模型进行一次优化。这样做一方面可以避免在优化判别模型的时候出现过拟合，另一方面可以避免生成模型因更新次数过多而导致收敛缓慢的问题。

一般来说，在 GAN 的训练过程中，初始状态的生成模型和判别模型都比较弱，主要通过交替训练来逐步共同提高。而且，我们应尽量避免训练中出现一个模型很强而另一个模型很弱的情形，因为这样会导致训练的优化方向出问题。一个比较强的模型不知道自己在和一个比较弱的模型博弈，而把对方也当成一个比较强的模型从而使得自己去过度拟合对方，最后导致训练收敛缓慢甚至训练失败。

在多数情况下，生成模型会比较弱，而判别模型会比较强。所以，我们可以考虑对生成模型进行一定程度的预训练（pre-training）以得到一个相对好一点的初始状态，然后可以有意地选取一些相对弱的判别模型来防止其快速收敛变强。

10.1.4 GAN 的扩展模型

在经典的生成对抗网络中，生成模型和判别模型都选用了全连接神经网络。这种结构比较适合用于一些简单的图像分类任务（比如手写数字识别）。而对于

较复杂的图像分类任务，我们一般会选择各种卷积神经网络来构建生成模型和判别模型。在这一节，我们将介绍 GAN 的几种常见的扩展模型。

DCGAN

在深度卷积 GAN（Deep Convolutional GAN，DCGAN）中，生成模型和判别模型都是深度卷积神经网络。在训练过程中，它的卷积计算和采样方式都有一些创新，以提升生成模型的表现能力。比如，在 DCGAN 的训练中，采用了带步长的卷积（strided convolution）和小步长卷积（fractionally-strided convolution），并学习空间下采样和上采样算子来更好地学习高维图像空间到低维特征表示空间之间的变换。

InfoGAN

信息 GAN（Information-theoretic GAN，InfoGAN）试图让 GAN 学到一些有意义可解释的特征表达方式，比如人脸识别中眼睛颜色、人物发型等特征。这种学习是完全无监督进行的，在手写数字识别数据集上，InforGAN 成功地分别学会了书写风格和数字形状。

CGAN

条件 GAN（Conditional GAN，CGAN）为 GAN 增加了限制条件，从而增加 GAN 的准确率。在 GAN 中，生成模型以一个噪声向量作为输入来生成假样本数据，导致其自由度非常大，从而很难收敛并产生高质量的假样本数据。CGAN 通过对生成模型和判别模型添加限制条件去解决上述难题。假设限制条件信息用变量y来表示，那么 CGAN 的价值函数$V(G,D)$可以写成如下形式：

$$\min_G \max_D V(G,D) = E_{x\sim p_{\text{data}}(x)}[\log D(x|y)] + E_{z\sim p_z(z)}\left[\log\left(1 - D(G(z|y))\right)\right]$$

SeqGAN

在序列 GAN（Sequence GAN，SeqGAN）中，生成模型是一个强化学习模型，而判别模型的结果被用作强化学习模型中的奖赏。这样，我们就可以用 GAN

的框架来进行强化学习了。

10.2 应用场景：搜索广告中由查询词直接生成关键词

在第 9 章提到了搜索广告中的查询词和关键词的匹配问题，并阐述了如何用词向量的方法来进行查询词和关键词的智能匹配。在了解了生成对抗网络之后，我们很自然地会考虑这样一个问题：能否训练一个条件生成对抗网络，使得其中的生成模型在给定查询词的条件下能够自动生成关键词？答案是肯定的。本节就对这一应用场景做具体介绍。

10.2.1 生成模型的构建

在这个应用场景里面，由于我们希望通过查询词直接生成关键词，那么很自然地就会想到序列到序列（seq2seq）模型。这是因为查询词和关键词都是包含多个单词的文本序列，从而我们可以把查询词作为输入的序列，把关键词作为输出的序列。考虑到经典的生成对抗网络中的生成模型一般以一个噪声向量作为输入，我们可以运用条件生成对抗网络并把查询词作为限制条件信息。具体来说，假定查询词$\boldsymbol{q}$包含T个单词$\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_T$，关键词$\boldsymbol{k}$包含$T'$个单词$\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_{T'}$，这其中的每个单词都可以用词向量的方式来表示。在编码（encoding）过程中，我们可以使用标准的递归神经网络（Recurrent Neural Network，RNN）来把查询词编码成为一个编码向量（thought vector），如下所示：

$$\boldsymbol{h}_t = \sigma_h(\boldsymbol{W}_h \boldsymbol{q}_t + \boldsymbol{U}_h \boldsymbol{h}_{t-1})$$

其中，$\boldsymbol{h}_t$是递归神经网络的隐藏层向量，$\boldsymbol{W}_h$和$\boldsymbol{U}_h$是参数矩阵，$\sigma_h(\cdot)$是激活函数。我们假设生成模型的输入噪声向量是$\boldsymbol{z}$，那么如图 10.2 所示，我们可以把查询词$\boldsymbol{q}$作为条件生成对抗网络中的限制条件信息，并使用一个常见的平行网络结构来把$\boldsymbol{z}$和$\boldsymbol{q}$的编码向量$\boldsymbol{h}_t$进一步编码成为向量$\boldsymbol{s}$，如下所示：

$$\boldsymbol{s} = \sigma_z(\boldsymbol{U}_h\boldsymbol{h}_t + \boldsymbol{U}_z\boldsymbol{z})$$

其中$\boldsymbol{U}_z$是参数矩阵，$\sigma_z(\cdot)$是激活函数。

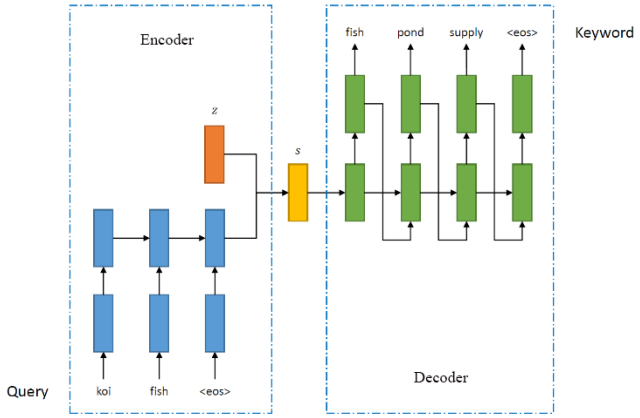


图 10.2　生成模型：序列到序列模型

生成模型的解码过程也是由递归神经网络来完成的，其目标是估计条件概率$p(\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_{T'}|\boldsymbol{z}, \boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_T)$。估计的过程是运用标准的语言模型（language model）从编码向量$\boldsymbol{s}$开始进行解码，如下式所述。即，解码过程在$\boldsymbol{t}$时刻的隐藏状态是$\boldsymbol{s}_t$，而我们令$\boldsymbol{s}_0 = \boldsymbol{s}$，那么：

$$p(\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_{T'}|\boldsymbol{z}, \boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_T) = \prod_{t=1}^{T'} p(\boldsymbol{k}_t|\boldsymbol{s}_{t-1}, \boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_{t-1})$$

如果我们将上式进一步简化为一阶语言模型，则可以把生成模型写成如下形式：

$$\begin{aligned}\boldsymbol{k} = G(\boldsymbol{z}|\boldsymbol{q}) &\propto p(\boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_{T'}|\boldsymbol{z}, \boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_T) \\ &= \prod_{t=1}^{T'} p(\boldsymbol{k}_t|\boldsymbol{s}_{t-1}, \boldsymbol{k}_1, \boldsymbol{k}_2, \ldots, \boldsymbol{k}_{t-1}) \\ &= \prod_{t=1}^{T'} p(\boldsymbol{k}_t|\boldsymbol{s}_{t-1}, \boldsymbol{k}_{t-1})\end{aligned}$$

$$\boldsymbol{s}_t = g(\boldsymbol{s}_{t-1}, \boldsymbol{k}_t, \boldsymbol{c}_t)$$

其中g是某种结构的递归神经网络，比如 LSTM 或者 GRU，$\boldsymbol{c}_t$是用来增强递归神经网络的注意力机制的向量（关于注意力模型，请读者参阅第 6 章）。

10.2.2 判别模型的构建

在这个场景里面，给定一对查询词$\boldsymbol{q}$和关键词$\boldsymbol{k}$，判别模型应该可以判断与$\boldsymbol{k}$相关联的广告被展现在$\boldsymbol{q}$的查询结果页面的时候，用户会不会点击这个广告。所以，判别模型的输入是查询词$\boldsymbol{q}$和关键词$\boldsymbol{k}$，输出是用户会点击（记为 1）或者不会点击（记为 0）。为此，我们可以考虑使用图 10.3 所示的平行递归神经网络构建判别模型。

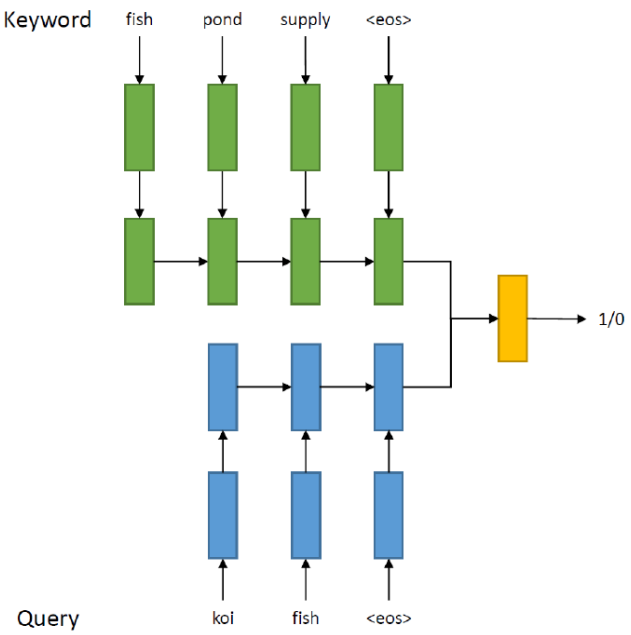


图 10.3 判别模型：平行的递归神经网络

10.2.3 条件生成对抗网络的构建

至此，我们有了生成模型$G(\boldsymbol{z}|\boldsymbol{q};\theta_g)$和判别模型$D(\boldsymbol{k}|\boldsymbol{q};\theta_d)$，其中$\theta_g$和$\theta_d$都是模型参数，那么我们就可以写出条件生成对抗网络的目标函数了。

$$\min_G \max_D V(G,D) = E_{(\boldsymbol{q},\boldsymbol{k})\sim p_{\text{data}}(\boldsymbol{q},\boldsymbol{k})}[\log D(\boldsymbol{k}|\boldsymbol{q})] + E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{q})|\boldsymbol{q}))]$$

目标函数由两部分组成，分别表示判别模型和生成模型的优化目标。仔细观察这一目标函数我们发现，它可以区分真实的$(\boldsymbol{q},\boldsymbol{k})$对和生成模型生成的假$(\boldsymbol{q},\boldsymbol{k})$对，但是它却无法区分有广告点击的真实$(\boldsymbol{q},\boldsymbol{k})$对和没有广告点击的真实$(\boldsymbol{q},\boldsymbol{k})$对。为了解决这一问题，需要再添加一个优化目标到目标函数中，并把目标函数最终写成如下形式：

$$\begin{aligned}\min_G \max_D V(G,D) = {} & E_{(\boldsymbol{q},\boldsymbol{k})\sim p_{\text{data}}(\boldsymbol{q},\boldsymbol{k})}[\log D(\boldsymbol{k}|\boldsymbol{q})] + E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z}|\boldsymbol{q})|\boldsymbol{q}))] \\ & + E_{(\boldsymbol{q},\boldsymbol{k})\sim p_{\text{data}}(\boldsymbol{q},\boldsymbol{k})}\left[\log\left(1 - D(\widehat{\boldsymbol{k}}|\boldsymbol{q})\right)\right]\end{aligned}$$

其中，$\widehat{\boldsymbol{k}}$表示用户提交查询词$\boldsymbol{q}$之后，并没有发生广告点击的真实$(\boldsymbol{q},\widehat{\boldsymbol{k}})$对。由此可见，判别模型不仅会把生成模型输出的假关键词判定为 0，还会把生成模型输出的没有发生广告点击的真实关键词也判定为 0。这样，只有当生成模型输出有广告点击的真实关键词的时候，判别模型才会判定为 1。当条件生成对抗网络训练收敛的时候，其中的生成模型就可以在给定查询词$\boldsymbol{q}$的条件下产生有广告点击的真实关键词$\boldsymbol{k}$了，这样我们就可以直接把这个生成模型用于备选广告选择了。图 10.4 展示了用于广告选择的条件生成对抗网络的整体网络结构。

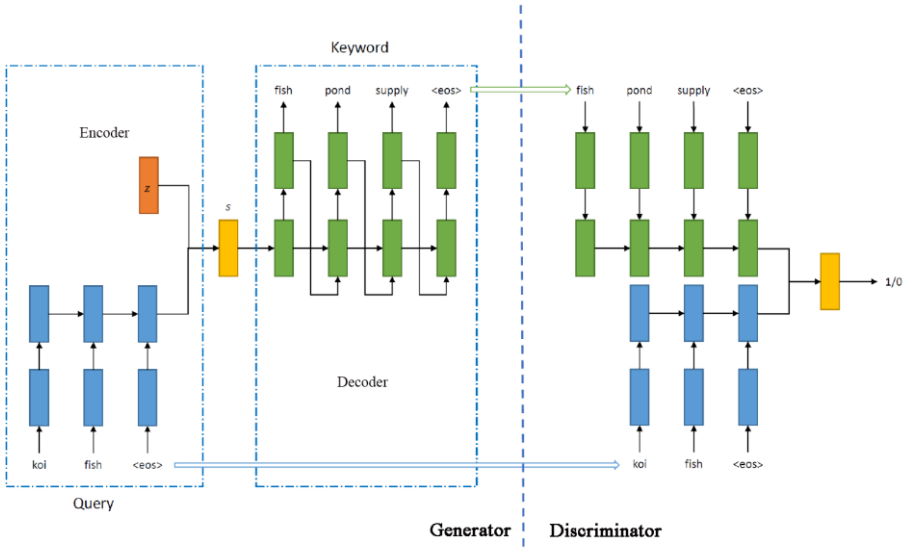


图 10.4　用于广告选择的条件生成对抗网络

10.3　本章小结

本章主要介绍了以下内容：

（1）生成对抗网络，主要讲解它的原理、构成、优化过程等。

（2）经典的生成对抗网络模型，以及几个扩展模型，比如条件生成对抗网络等。

（3）以搜索广告中由查询词直接生成关键词的问题为例，讲解条件生成对抗网络模型的应用。

生成对抗网络的提出引领了一场研究新浪潮，在它被提出后的三四年内，已有几百篇相关的研究论文发表。有人把这一现象总结出了一个叫作 The GAN Zoo 的项目放在 Github 上，可见其影响力之大。与此同时，生成对抗网络自身仍有很多问题待解决，比如在训练中比较普遍的生成模型崩溃（generator collapsing）问题，即生成模型只能输出一小类和真实数据相似的样本（部分崩

溃），甚至无法生成和真实数据相似的样本（完全崩溃）。再比如训练的不稳定性，结果只能收敛到鞍点而不是局部极小值。这些问题都需要研究者进行更加大量深入的工作来进行改进。

参考文献

[1] GOODFELLOW I J, SHLENS J, SZEGEDY C. Explaining and harnessing adversarial examples, arXiv preprint arXiv:1412.6572. 2014.
[2] GOODFELLOW I J, POUGET-ABADIE J, MIRZA M, et al, Generative adversarial nets. Advances in neural information processing systems 2014[C],2672-2680.
[3] GOODFELLOW I J, POUGET-ABADIE J, MIRZA M, et al, Generative adversarial nets. Advances in neural information processing systems 2014[C],2672-2680.
[4] MIRZA M, OSINDERO S. Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784,2014.
[5] CHEN X, DUAN Y, HOUTHOOFT R, et al, Infogan: Interpretable representation learning by information maximizing generative adversarial nets. Advances in neural information processing systems 2016[C] , 2172-2180.
[6] RADFORD A, METZ L, CHINTALA S, Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434,2015.
[7] LEE M C, GAO B, ZHANG R. Rare query expansion through generative adversarial networks in search advertising. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining 2018[C]. ACM,2018.

Deep Learning Model and Application

第 11 章
深度强化学习

深度强化学习（Deep Reinforcement Learning，又称深度增强学习）是将深度学习与强化学习相结合，从而实现从感知到决策的一种机器学习算法框架。深度强化学习的优势在于其具备完全自主的学习某种技能的能力，它使得计算机可以通过对外界反馈进行自我调整的自学方式来掌握某种技能，甚至超过人类的水平。Google DeepMind 推出的智能围棋程序 AlphaGo 就是通过深度强化学习算法来自我练就成可以战胜人类围棋大师的高手的。本章将介绍深度强化学习的原理及其在推荐系统中的应用。

11.1 深度强化学习的原理

11.1.1 强化学习的基本概念

在介绍深度强化学习之前，我们需要对强化学习有一个大致的了解。

在人工智能领域，一般用智能体（agent）来表示一个具备某种智能的实体，比如机械手、无人车、电脑棋手、人等。强化学习研究的就是如何通过智能体（agent）和环境（environment）之间的交互来学习某种技能的问题。

比如，机械手要从地上抓起一个零件放在箱子上，那么机械手周围的物体包括地面、零件、箱子就都是环境，机械手通过自身的摄像头或者传感器来感知环境（机械手与各种物体的相对位置等），然后通过一系列的动作来完成抓起零件放在箱子上这个任务。

再比如，我们玩切水果的手机游戏时看到的屏幕就是环境，通过一些动作（不同方向的触屏滑动操作）来完成切碎水果的任务。

不管是什么样的任务，都包含了一系列的环境观察（observation）、动作（action）和反馈（reward，或者叫奖赏）。这里用观察来表示智能体对环境的感知，是因为智能体不一定能够完全感知环境信息，比如机械手上的摄像头只能得到某几个角度的画面。所谓反馈是指，智能体在观察了环境并根据环境采取了某

个动作之后，环境的变化所造成的某种影响。比如，当机械手距离零件更近时，就会得到正反馈；当机械手触碰地面被卡住时，就会得到负反馈；当我们成功地切碎了几个水果时，就会得到几百分的奖励；当我们不巧切到了手雷时，就会直接输掉游戏。图 11.1 表示了这样一个循环交互过程。

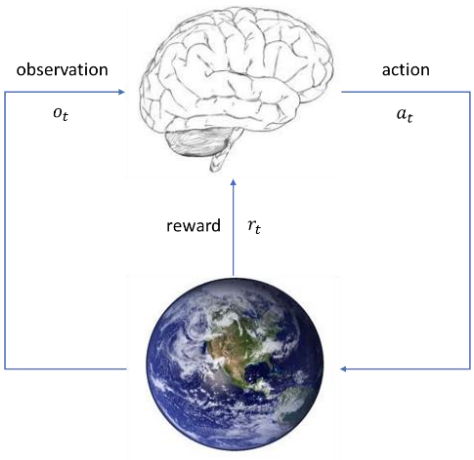


图 11.1　智能体与环境的交互

其实这也就是人与环境之间交互的一种模型刻画。在某个时刻t，智能体对环境有了一个观察o_t，根据这个观察智能体选择执行某个动作a_t，这个动作会对环境产生一定的影响使得环境发生变化，而这个影响会以奖赏r_t的形式反馈给智能体。这个过程循环进行，智能体的目标就是要尽可能多地获得奖赏。

一般来讲，在每一个时刻，智能体都会根据其当前的状态（state）s_t来决定它要采取的动作a_t，状态s_t可能不仅包含当前的观察o_t，还包含此前几个时刻的观察及一些其他的辅助信息。即，状态s_t和动作a_t之间存在着某种映射关系，一个状态对应着一个动作，或者一个状态对应采取不同动作的概率。这种映射关系，称之为策略（policy）π，记为映射函数$a = \pi(s)$或者概率表示$\pi(a|s)$。强化学习的目标就是要找到能够最大化奖赏的最优策略。

强化学习的训练过程一般是从随机策略开始的。根据随机策略，我们可以得

到一系列的状态、动作和反馈：

$$\{s_1, a_1, r_1, s_2, a_2, r_2, \ldots, s_t, a_t, r_t\}$$

这就构成了训练样本，强化学习根据这些训练样本来优化调整策略，从而使策略在训练样本上得到最优的奖赏。这个过程和人在玩游戏时一般会越玩越好是一样的。比如在切水果的游戏中，我们根据反馈会调整动作尽可能多地切到水果而且避免切到手雷。

11.1.2 马尔可夫决策过程

在了解了强化学习的基本概念以后，接下来介绍强化学习里面常用的马尔可夫决策过程（Markov Decision Process，MDP）。

MDP 是基于一个马尔可夫性的假设：未来只取决于当前，而与过去无关。即一个事物未来会变成什么样子，只和其当前的状态有关系，而与其过去是什么样子没有关系。从数学的角度而言，当且仅当下式满足的时候，我们就称状态s_t具备马尔可夫性：

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_t, s_{t-1}, \ldots, s_0)$$

这是一个典型的一阶马尔可夫决策过程。

有了马尔可夫性的假设之后，马尔可夫决策过程就可以用(S, A, P)来表示，其中S表示状态，A表示动作，P表示状态之间的转移概率，即采取动作a_t之后从状态s_t转移到状态s_{t+1}的概率。如果我们知道了所有的状态转移概率，就可以轻松地通过在每个时刻选取最优的动作来保证获得最大化的奖赏。但是在大多数实际情况下，我们是很难知道所有的状态转移概率的，而是需要一些特定的算法来寻找最优策略。

11.1.3 价值函数和贝尔曼方程

马尔可夫决策过程的每一个状态s_t对应一个动作a_t，这个动作会带来奖赏r_t，除这个即时奖赏外，为了衡量状态s_t的好坏，还需要定义状态的价值函数。有的时候，动作a_t带来的奖赏r_t可能不高，但是它使得 MDP 转移到了一个更好的状态s_{t+1}，从s_{t+1}开始以后的累积奖赏可能非常丰厚。因而，状态的好坏等价于对未来累积奖赏的期望。从t时刻开始到未来的累积奖赏可以写为：

$$G_t = r_{t+1} + \lambda r_{t+2} + \cdots = \sum_{k=0}^{\infty} \lambda^k r_{t+k+1}$$

这里$\lambda < 1$是衰减因子，表示当前奖赏的权重是比较大的，未来时间越久的奖赏权重越小。那么，状态的价值函数（value function）$v(s)$就是对其未来累积奖赏取期望：

$$v(s) = E[G_t | s_t = s].$$

此前，我们只能通过直接优化策略$a = \pi(s)$或者$\pi(a|s)$来寻找最优策略π。有了价值函数之后，就可以通过估计价值函数来间接寻找最优策略了，这是因为价值函数反映了状态的好坏，而我们一般倾向于选择可以转移到好状态的策略。

接下来的问题是如何对价值函数进行估值，这就需要引入贝尔曼方程（Bellman equation）。首先把价值函数展开：

$$\begin{aligned} v(s) &= E[G_t | s_t = s] \\ &= E[r_{t+1} + \lambda r_{t+2} + \lambda^2 r_{t+3} + \cdots | s_t = s] \\ &= E[r_{t+1} + \lambda(r_{t+2} + \lambda r_{t+3} + \cdots) | s_t = s] \\ &= E[r_{t+1} + \lambda G_{t+1} | s_t = s] \\ &= E[r_{t+1} + \lambda v(s_{t+1}) | s_t = s] \end{aligned}$$

于是就得到了下面这个被称为贝尔曼方程的数学公式：

$$v(s) = E[r_{t+1} + \lambda v(s_{t+1}) | s_t = s]$$

可以看出，当前状态的价值与下一状态的价值、当前的奖赏有关，因而当前状态的价值是可以通过迭代来进行计算的。

在寻找最优策略的时候，不仅要衡量一个状态的价值，还要看在这个状态下采取某个动作的价值，然后选择价值最大的那个动作向前推进。因而我们需要继续定义动作价值函数（action value function），如下：

$$\begin{aligned} q^{\pi}(s,a) &= E[r_{t+1} + \lambda r_{t+2} + \lambda^2 r_{t+3} + \cdots | s,a] \\ &= E_{s'}[r + \lambda q^{\pi}(s',a') | s,a] \end{aligned}$$

动作价值函数比状态价值函数更加直观地反映了在策略π下每个动作的价值，因而我们在寻找最优策略的算法中更多地会使用动作价值函数。

求解 MDP 最优策略的方法有三种：

第一种是基于策略（policy-based）的方法，就是直接计算策略函数；

第二种是基于模型（model-based）的方法，就是估计出状态转移函数；

第三种是基于价值（value-based）的方法，也就是找到最优的动作价值函数。

本节重点介绍基于价值的方法。

最优的动作价值函数就是所有可能策略下动作价值函数的最大值，其贝尔曼方程可以写作：

$$\begin{aligned} q^*(s,a) &= \max_{\pi} q^{\pi}(s,a) \\ &= E_{s'}\left[r + \lambda \max_{a'} q^*(s',a') | s,a\right] \end{aligned}$$

从等式右侧可以看出，最优策略会选择使得动作价值函数取得最大值的动作a'。类似地，我们也可以写出最优价值函数的贝尔曼方程：

$$v_*(s) = \max_{a} E[r_{t+1} + \gamma v_*(s_{t+1}) | s_t = a, a_t = a]$$

11.1.4 策略迭代和值迭代

策略迭代和值迭代是基于贝尔曼方程求解最优策略的两个基本方法。本节以状态价值函数为例进行介绍，动作价值函数的迭代公式与之类似。

1. 策略迭代（policy iteration）

策略迭代是通过贝尔曼方程直接迭代计算价值函数，从而使策略收敛到最优。根据状态价值函数的贝尔曼方程，可以写出下面从第k步到第$k+1$步的迭代公式：

$$\begin{aligned} v_{k+1}(s) &= E[r_{t+1} + \lambda v_k(s_{t+1}) | s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \end{aligned}$$

策略迭代的基本过程是：首先使用当前策略产生一些新的样本，接着用这些新的样本去更好地估计策略的价值，然后用策略的价值更新策略。这样的过程反复进行，直到找到最优策略。具体可以总结为算法 11.1。

算法 11.1　策略迭代算法

（1）初始化

对所有的 $s \in S$，将$v(s) \in R$ 和 $\pi(s) \in A(s)$随机初始化。

（2）策略评估

重复

$\Delta \leftarrow 0$

对每一个 $s \in S$：

$$\bar{v} \leftarrow v(s)$$

$$v(s) \leftarrow \sum_{s',r} p\big(s', r|s, \pi(s)\big)[r + \gamma v(s')]$$

$$\Delta \leftarrow \max(\Delta, |\bar{v} - v(s)|)$$

直到$\Delta < \theta$（θ是一个很小的正实数）

（3）策略改进

$$\text{policystable} \leftarrow \text{true}$$

对每一个 $s \in S$:

$a \leftarrow \pi(s)$

$\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$

如果 $a \neq \pi(s)$，那么 policystable ← false。

如果 policystable为true，那么算法终止并输出v和π；否则执行第（2）步。

可以看出，策略迭代分为两阶段：

在策略评估（policy evaluation）阶段，更新状态价值函数，从而更好地估计当前策略的价值。

在策略改进（policy improvement）阶段，运用贪心法产生新的样本用于下一轮的策略评估。

另外，我们注意到策略迭代的算法是需要用到状态转移概率p的，而且在理想的情况下是需要遍历所有状态的。

2. 值迭代（value iteration）

值迭代是通过贝尔曼最优方程直接迭代计算价值函数，从而使策略收敛到最优。根据状态价值函数的贝尔曼方程，我们可以写出下面从第k步到第$k+1$步的迭代公式：

$$\begin{aligned} v_{k+1}(s) &= \max_a E[r_{t+1} + \gamma v_k(s_{t+1}) | s_t = a, a_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \end{aligned}$$

值迭代的基本过程是使用贝尔曼最优方程来直接更新价值函数，最后得到的价值v_*就是当前状态下的最优价值，因而相应的策略就是最优策略。具体可以总结为算法 11.2。

算法 11.2　值迭代算法

（1）初始化

对所有的 $s \in S$，令$v(s) = 0$。

（2）值更新

重复

$\Delta \leftarrow 0$

对每一个 $s \in S$。

$$\bar{v} \leftarrow v(s)$$

$$v(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$$

$$\Delta \leftarrow \max(\Delta, |\bar{v} - v(s)|)$$

直到$\Delta < \theta$（θ是一个很小的正实数）。

（3）输出确定性的策略 π，使得

$$\pi(s) \leftarrow \operatorname*{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$$

我们注意到，值迭代的算法也是需要用到状态转移概率p的，适用于状态转移概率已知的简单问题。

11.1.5　Q-Learning

此前我们曾经提到，动作价值函数比状态价值函数可以更加直观地反映在策略π下每个动作的价值，因而我们在寻找最优策略的算法中更多地会使用动作价值函数。在 11.1.4 节我们知道，值迭代使用贝尔曼最优方程来直接更新价值函数，比策略迭代更简洁。那么我们就考虑一下在值迭代算法下利用动作价值函数的贝尔曼最优方程的情况。在这种情况下，根据动作价值函数的贝尔曼最优方程，值迭代的公式可以写为：

$$q_{k+1}(s, a) = E_{s'}\left[r + \lambda \max_{a'} q_k(s', a') \,|s, a\right]$$

可以看出，每次迭代都需要对所有的状态和动作的q值更新一遍。在实际问题中，通常只能得到有限的样本，而不可能遍历所有的状态和动作。为了解决这一难题，Q-Learning 提出了更新q值的一种新方法：

$$q(s_t,a_t)\leftarrow q(s_t,a_t)+\alpha\left(r_{t+1}+\lambda\max_{a'}q(s_{t+1},a')-q(s_t,a_t)\right)$$

可以看出，在用值迭代的方式计算出动作价值函数的估值$r_{t+1}+\lambda\max_{a'}q(s_{t+1},a')$后，新算法并没有把这个估值直接赋给新的$q(s_t,a_t)$，而是计算了估值与旧的$q(s_t,a_t)$的误差，并把这个误差以类似梯度更新的方式来更新$q(s_t,a_t)$值，$\alpha$用来控制梯度更新的大小。这样做的好处是，可以使用有限的样本来计算一些梯度，然后用这些梯度来更新所有的q值，用类似随机梯度下降的方式来收敛到最优的q值，从而得到最优策略。Q-Learning 的具体算法见算法 11.3。

算法 11.3　Q-Learning 算法

（1）初始化

对所有的 $s\in S$和$a\in A(s)$，将$q(s,a)$随机初始化；并且令$q(\tilde{s},\cdot)=0$，其中$\tilde{s}$为终止状态。

（2）q值更新

对于每一个样本序列，重复初始化状态s。

对于样本序列中的每一步，重复使用“某个策略”，根据状态s选择一个动作a'来执行。

动作执行完毕，得到奖赏r'和新的状态s'：

$$q(s,a)\leftarrow q(s,a)+\alpha\left(r'+\lambda\max_{a'}q(s',a')-q(s,a)\right)$$

$$s\leftarrow s'$$

直到s到达$\tilde{s}$。

我们可以看到，在 Q-Learning 算法中需要使用“某个策略”来选择动作，这个策略并不是目标策略，所以 Q-Learning 算法是 off-policy 的算法。那么这

个策略怎么选择呢？一种选择是使用随机策略，这种策略的好处是：可以探索（exploration）未知动作的价值，从而有机会找到更好的目标策略；另一种选择是根据当前的q值计算出一个最优的动作，即$\pi(s_{t+1}) = \underset{a}{\operatorname{argmax}}\, q(s_{t+1}, a)$，这种策略称为贪心策略（greedy policy），是一种开发（exploitation）的方式，有利于随时验证当前策略是否有效。当然，还可以把上面两种方式结合起来综合运用，就有了所谓的ϵ-greedy 策略，即以ϵ的概率选择随机策略，以$1-\epsilon$的概率选择贪心策略，其中ϵ是个很小的值。这样，就可以尽可能地平衡探索和利用两种策略（exploration-exploitation tradeoff）。

11.1.6 深度 Q 网络

到目前为止，我们已经把强化学习的基本概念和原理做了简单的介绍。接下来，要介绍深度学习与强化学习相结合的重要算法深度 Q 网络（Deep Q-Network，DQN）模型。

在 11.1.5 节我们看到，寻找最优策略的关键步骤是估计和更新q值$q(s,a)$。在状态不多动作也不多的情况下，可以用矩阵来存储和计算$q(s,a)$。但是在现实的问题中，有时状态空间是非常巨大的。以赛车类电子游戏为例，我们一般可以把屏幕上每一秒的图像截屏作为状态。即使是在低分辨率（480 像素 × 360 像素）、低色彩度（每像素 256 种颜色）的情况下，状态的数量就有$256^{480\times360}$之多。有时动作空间也是非常巨大的，比如我们将在 11.2 节介绍的推荐系统，可供推荐的物品组合数也是一个天文数字。因而，不可能用矩阵来存储和计算$q(s,a)$。

为了解决这个问题，需要引入一个函数$f(s,a)$来近似计算$q(s,a)$，即：

$$q(s,a) \approx f(s,a)$$

我们把$f(s,a)$称为价值函数近似（value function approximation），它可以是任意类型的函数。假定它的参数表示为w，就有：

$$q(s,a) \approx f(s,a,w)$$

当状态空间比较大而动作空间比较小时，可以把上面的近似方程简化为：

$$q(s) \approx f(s,w)$$

它的输入是一个状态s，输出是一个含有所有动作的q值的向量$\left(q(s,a_1),q(s,a_2),\ldots,q(s,a_n)\right)^{\mathrm{T}}$，其中$n$是动作的数量。比如赛车类电子游戏中$n$=4，对应于加速、减速、左转、右转 4 个动作。

接下来的问题就是如何表示和计算这个价值函数近似$f(s,a,w)$，这就轮到深度学习出场了。我们知道，深度神经网络可以用来模拟$f(s,a,w)$。以键盘手柄操作的电子游戏为例，其 DQN 结构如图 11.2 所示。它的输入可以是连续几帧的屏幕图像，先经过几个卷积层，再经过几个全连接层，最后输出含有所有动作的q值的向量。深度学习的引入，使得我们很容易将大规模状态空间的q值计算问题通过设计 DQN 来实现。

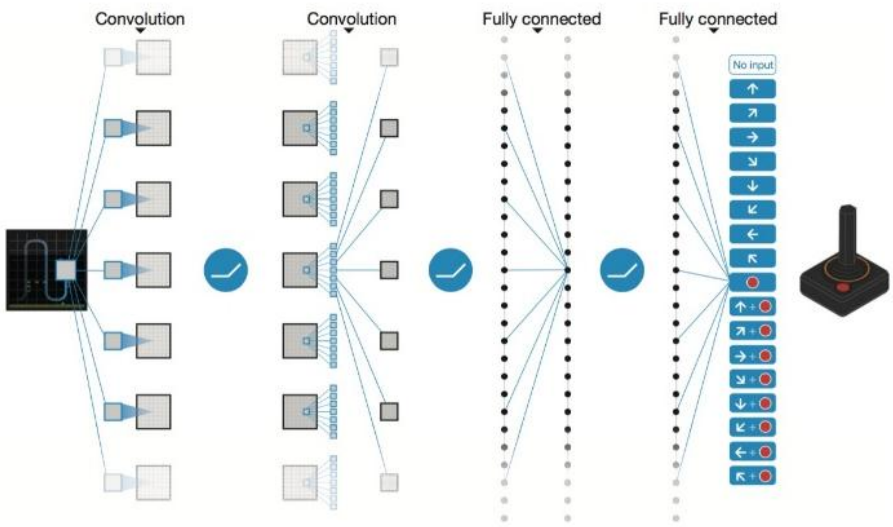


图 11.2　电子游戏的 DQN 示例

在搭建好 DQN 以后，就要考虑怎样训练这样一个神经网络。DQN 的输入是状态，输出是在输入状态下每个动作的q值。对于 DQN 输出端的监督信息，可以参考 11.1.5 节介绍的 Q-Learning 算法计算出来。在 Q-Learning 算法里，q值更新时的估值是$r_{t+1}+\lambda \max_{a'} q(s_{t+1},a')$，它恰好可以作为 DQN 输出端$q$值的监督信息来计算误差。因而，DQN 的损失函数就很自然地定义为下面的均方误差期望的形式：

$$L(w)=E\left[\left(r+\gamma \max_{a'} q(s',a',w)-q(s,a,w)\right)^2\right]$$

其梯度为：

$$\nabla_{w_i} L(w_i)=E\left[\left(r+\gamma \max_{a'} q(s',a',w_{i-1})-q(s,a,w_i)\right)\nabla_{w_i} q(s,a,w_i)\right]$$

经典的 DQN 算法流程如算法 11.4 所示。

算法 11.4　DQN 算法流程

（1）初始化重播记忆（replay memory）D的容量为N。

（2）随机初始化动作价值函数q的参数w。

（3）对于每一个样本序列，重复以下内容：

读取状态s_1，对$t=1,\cdots,T$执行；

以概率ϵ选择一个随机动作a_t；

否则选择动作$a_t=\max_a q^*(s_t,a,w)$；

执行动作a_t，从而获得奖赏r_t并到达状态s_{t+1}；

将状态转移(s_t,a_t,r_t,s_{t+1})存储于D。

从D中随机抽取一小批量（minibatch）状态转移数据(s_j,a_j,r_j,s_{j+1})。

设置$y_j=\begin{cases} r_j & \text{当} s_{j+1}\text{是终止状态时} \\ r_j+\gamma \max_{a'} q(s_{j+1},a',w) & \text{当} s_{j+1}\text{不是终止状态时} \end{cases}$

进行一次梯度下降操作 $\nabla_{w_i} L(w_i)=E\left[\left(y_j-q(s_j,a_j,w_i)\right)\nabla_{w_i} q(s_j,a_j,w_i)\right]$

上述 DQN 算法中涉及了经验池（experience replay）技巧。在训练 DQN 时，采集的样本序列一般都是时间序列，样本之间有一定的连续性，实验表明在更新q值时效果会不好。为此，我们可以先把状态转移样本(s_t, a_t, r_t, s_{t+1})存储到重播记忆（replay memory）D 里，然后再从 D 中随机抽样来进行训练。这就是所谓的经验池技巧。总之，DQN 的训练就是先通过 Q-Learning 获取大量的训练样本(s_t, a_t, r_t, s_{t+1})，然后对神经网络进行随机梯度下降训练。

11.1.7 策略梯度

DQN 算法是一种基于价值的方法，它先估计每一个状态动作的价值，然后选取具有最大价值的动作来执行，这是一种间接的方法。此外，还可以设计一种模型根据输入的状态直接选取动作来执行，这就是所谓的策略网络（policy network）模型。

策略网络π一般是一个深度神经网络，它的输入是状态s，输出是动作a，网络参数是θ。它可以写成函数表达$a = \pi(s, \theta)$或者概率表达$a = \pi(a|s, \theta)$。训练出了策略网络，也就很自然地找到了最优策略。那么接下来的问题就是如何定义策略网络的目标函数$L(\theta)$，以及拿什么来作为监督信息。

对于监督信息，既然我们要评价策略网络输出动作的好坏，那就不妨假设有一个函数$f(s, a)$可以评价给定状态s下输出动作a的好坏，函数值越大表示动作越好。对于好的动作，我们就希望增加输出这个动作的概率；对于坏的动作，我们就希望减少输出这个动作的概率。要达到这种目的，我们的目标就可以是极大化：

$$E[f(s,a)] = \sum_{s,a} \pi(a|s,\theta) f(s,a)$$

或者写成极小化：

$$L(\theta) = -\sum_{s,a} \pi(a|s,\theta) f(s,a)$$

这样，就可以计算出目标函数的梯度$\nabla_\theta L(\theta)$，并使用梯度下降的方式来进行优化。这也就是所谓的策略梯度（policy gradient）方法。梯度的计算如下：

$$\begin{aligned}\nabla_\theta L(\theta) &= -\nabla_\theta \sum_{s,a} \pi(a|s,\theta) f(s,a) \\ &= -\sum_{s,a} \nabla_\theta \pi(a|s,\theta) f(s,a) \\ &= -\sum_{s,a} \pi(a|s,\theta) \frac{\nabla_\theta \pi(a|s,\theta)}{\pi(a|s,\theta)} f(s,a) \\ &= -\sum_{s,a} \pi(a|s,\theta)\, \nabla_\theta \log \pi(a|s,\theta)\, f(s,a) \\ &= -E[f(s,a)\, \nabla_\theta \log \pi(a|s,\theta)]\end{aligned}$$

至此，关于策略梯度方法，就只剩下一个问题需要解决了，那就是如何定义评价函数$f(s,a)$。这里我们给出几种常用的选择。

（1）$\sum_{t=0}^{\infty} r_t$：从初始状态开始的累积奖赏。

（2）$\sum_{t'=t}^{\infty} r_{t'}$：从动作$a_t$开始的累积奖赏。

（3）$\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$：从动作$a_t$开始的累积奖赏，并以状态$s_t$为参照偏差。

（4）$q(s_t, a_t)$：动作价值函数。

（5）$A(s_t, a_t)$：优势函数（advantage function）。

（6）$r_t + v(s_{t+1}) - v(s_t)$：TD 残差（temporal difference residual）。

其中，$v(s_t) = E_{s_{t+1:\infty}, a_{t:\infty}}[\sum_{l=0}^{\infty} r_{t+l}]$，$q(s_t, a_t) = E_{s_{t+1:\infty}, a_{t+1:\infty}}[\sum_{l=0}^{\infty} r_{t+l}]$，$A(s_t, a_t) = q(s_t, a_t) - v(s_t)$。

11.1.8 动作评价网络

在 11.1.7 节我们注意到，策略梯度框架里面的动作评价函数可以是动作价值函数$q(s_t, a_t)$。即在这种情况下，除策略网络外，我们还需要一个 DQN 来计算q值为策略网络提供评价信息（即监督信息）。这种架构有一个特殊的名字叫作动作评价网络（actor critic），actor 就是策略网络，用来输出动作，而 critic 就是价

值网络，用来评价动作的好坏，一般使用 DQN。图 11.3 给出了动作评价网络的基本框架。

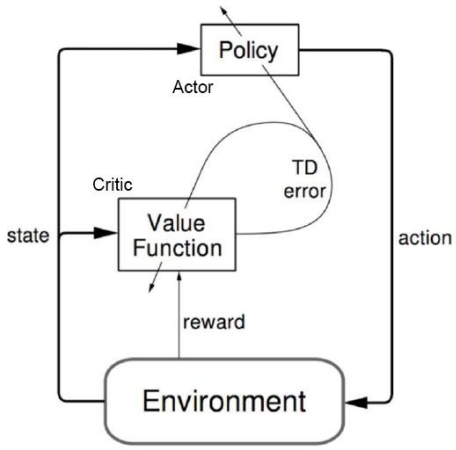


图 11.3　动作评价网络的基本框架

动作评价网络的运作方式非常类似于人类的学习方式。人类通过自己的思维方式来对外部环境采取某个动作，然后外部环境针对这一动作给出反馈。对这个反馈，人类依据自己的价值观予以评判来检验刚才那个动作的好坏，并相应地改进自己的思维方式。

11.2　应用场景：基于深度强化学习的推荐系统

推荐系统在电子商务领域有着广泛的应用，比如用户在网上商城查询了某种感兴趣的商品后，系统可以根据用户的兴趣和行为推荐一些用户可能感兴趣的其他商品。一方面，可以帮助用户快速找到他感兴趣的商品，从而提升用户的购物体验；另一方面，可以帮助商家完成更多的交易从而提高商品的销售量。

推荐系统的常用算法包括基于协同过滤（collaborative filtering）的方法、基于内容的方法、基于排序学习的方法等。这些方法把推荐看作一个静态的过程，

无法处理用户需求的动态变化。此外，这些方法的优化目标一般是某种短期利益，几乎不关注长期利益。

如果我们用深度强化学习来构建推荐系统，就可以解决上述问题。首先，强化学习是一个动态的学习过程，可以随时应对用户的兴趣变化。其次，强化学习可以通过对价值函数的定义来权衡短期利益和长期利益的关系。再次，深度强化学习借助深度学习的力量，能够处理推荐系统中庞大的状态空间和动作空间。

根据深度强化学习的框架，我们可以把推荐系统看作智能体，而把用户看作环境。推荐系统给用户推荐商品，用户的反馈（忽略、点击查看、购买）可以作为推荐系统的奖赏，随着用户和推荐系统的交互，这些奖赏可以不断累积。我们用马尔可夫决策过程来定义推荐系统。

- 状态空间S：状态$s_t \in S$定义为用户在时刻t之前的浏览记录，这些浏览记录按照时间排序。比如在t之前用户看到的m件商品，每件商品可以用标识序号（ID）来表示，也可以用商品描述（商品名称、属性、简介等）的词向量来表示。我们把这m件商品的表示向量拼在一起成为一个新向量来表示状态s_t。由于商品可能有数十万种甚至更多，m件商品的排列组合就是一个天文数字。
- 动作空间A：动作$a_t \in A$表示在t时刻推荐系统给用户的推荐。假如每次推荐系统给用户推荐一件商品，那么就可以用这件商品的标识序号向量或者商品描述向量来表示动作a_t。假如每次推荐n件商品，那么就可以把这n件商品的表示向量拼在一起成为一个新向量来作为动作a_t。可见，可选动作的数量也是非常大的。
- 奖赏R：当推荐系统根据状态s_t执行了动作a_t以后，用户就看到了系统给他推荐的商品。接下来，用户的行为（忽略、点击查看、购买）就可以作为奖赏反馈给推荐系统。比如，忽略、点击查看、购买这三种行为对系统的奖赏可以定义为 0、1、5。

- 转移概率P：转移概率$p(s_{t+1}|s_t,a_t)$表示在状态s_t下执行动作a_t时，状态转移到s_{t+1}的概率。根据马尔可夫性，我们知道$p(s_{t+1}|s_t,a_t,\dots,s_1,a_1)=p(s_{t+1}|s_t,a_t)$。
- 衰减因子γ：$\gamma\in[0,1]$表示我们对未来奖赏的衰减因子。当$\gamma=0$时，表示系统只考虑当前时刻的奖赏；当$\gamma=1$时，表示系统认为未来所有时刻的奖赏等同重要。

这样，我们就定义了用于推荐系统的马尔可夫决策过程(S,A,R,P,γ)，其目标是找到一个最优的推荐策略$\pi:S\to A$，使得推荐系统的累积奖赏可以最大化。

下面我们就要考虑模型的选择。

首先，由于推荐系统的状态空间和动作空间都很大，不可能用矩阵来存储和计算q值，所以我们想到使用经典的 DQN 模型，如图 11.4 所示。我们用 11.1.6 节介绍的算法来求解这个模型。

然后，我们还可以考虑使用动作评价网络框架，即把图 11.4 所示的 DQN 网络作为价值网络来计算q值，并且再加上一个输入为状态s_t输出为动作a_t的策略网络来优化策略π。

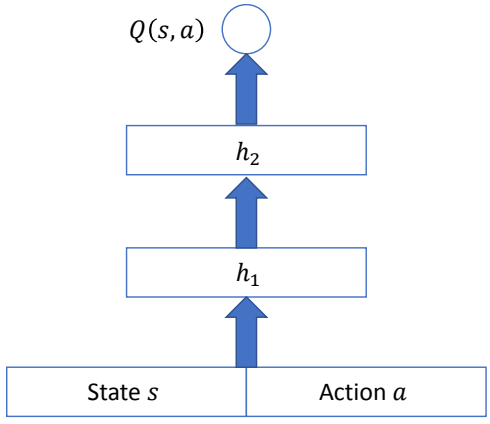


图 11.4　用于推荐系统的经典 DQN 模型

11.3 本章小结

本章介绍了强化学习及深度强化学习的基本概念、原理，以及核心算法DQN、策略梯度、动作评价网络等，并以推荐系统为应用背景介绍了深度强化学习的应用。

其实，深度强化学习的应用范围非常广泛，除我们比较熟悉的游戏领域外，它还可以用于排序学习、推荐系统、对话聊天系统、机器人控制等领域。有人甚至说“人工智能 = 深度学习 + 强化学习”，这更凸显了深度强化学习的重要性。

当然，深度强化学习的研究仍然处于起步阶段，还有许多问题待解决。比如，深度强化学习的样本利用率一般比较低，为了训练出好的模型，我们需要准备极为大量的样本。还有，好的深度强化学习模型离不开好的奖赏机制或者评价函数，而这种奖赏机制或者评价函数往往很难设计。此外，算法的不稳定性和对环境的过拟合也往往会导致训练失败。总之，这些问题还需要研究人员投入大量的精力来逐步解决。

参考文献

[1] SUTTON R S, BARTO A G. Reinforcement learning: An introduction. MIT press, 2018.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602, 2013.

[3] MNIH V, KAVUKCUOGLU K, SILVER D, et al, Human-level control through deep reinforcement learning. Nature [J] 518(7540):529, 2015.

[4] SCHULMAN J, MORITZ P, LEVINE S, et al, High-dimensional continuous control using generalized advantage estimation. arXiv preprint arXiv:1506.02438, 2015.

[5] LILLICRAP T P, HUNT J J, PRITZEL A, et al, Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971,2015.

[6] MNIH V, BADIA A P, MIRZA M, et al, Asynchronous methods for deep reinforcement learning. International conference on machine learning 2016[C],1928-1937.

[7] ZHAO X, ZHANG L, DING Z, et al, Recommendations with negative feedback via pairwise deep reinforcement learning. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining 2018[C]. ACM,1040-1048.

Deep Learning Model and Application

第 12 章

工程实践和线上优化

在之前的章节中，我们介绍了一些在实际项目中取得良好效果的模型。本章将讨论如何优化模型的处理速度，使得模型能够顺利上线。

在深度学习模型用于预测的任务中，在线预测是其中最为困难的一类任务之一。譬如在搜索任务中，如 Google 或者必应搜索，在线预测需要在几十毫秒甚至几毫秒中返回结果，这要求深度学习模型有很快的处理速度。本文将以序列到序列（Sequence to Sequence，Seq2Seq）模型为例，探讨对模型处理速度的优化。我们将 Seq2Seq 模型的处理性能从 100ms 优化到 3.6ms，成功应用于“必应”在线广告系统。接着，我们会以案例的形式介绍将性能优化的技术应用于 GBDT（Gradient Boosting Decision Tree）模型中，该成果发表在 KDD2018[1]。

12.1 Seq2Seq 模型介绍

Seq2Seq 模型是一种通用的编码器—解码器框架。Seq2Seq 模型主要是用来解决将一个序列 X 转化为另一个序列 Y 的一类问题，可用于机器翻译、文本摘要、会话建模、图像字幕等场景。在搜索广告任务中，Seq2Seq 模型可以用于用户在线查询的改写、广告关键字的改写等，从而提高召回率，覆盖更多的广告。

Seq2Seq 模型的结构为递归神经网络（RNN），其每个单元包含一个 word embedding 结构和 LSTM 单元。word embedding 即词嵌入向量，它对序列中的每个单词用一个对应的向量（往往是低维向量）来表示。单词所包含的信息存储在向量中，同时方便计算。LSTM（长短期记忆模型）由 Hochreiterh 和 Schmidhuber 于 1997 年推出，能够学习长距离的依赖关系。

图 12.1 展示了 Seq2Seq 模型应用于聊天场景的例子，用户收到信息“你在

1 YE T, ZHOU H C, ZOU W Y, et al, Rapidscorer: fast tree ensemble evaluation by maximizing compactness in data level parallelization. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining 2018[C], ACM, 941-950.

哪？”Seq2Seq 可以自动生成一些可能的回答，譬如中性的回答：“我在家”“我在公司”“我在深圳”；暖心的回答：“我在你心里”“我在你身旁守护你”；或者捣蛋的回答：“我不告诉你”等。

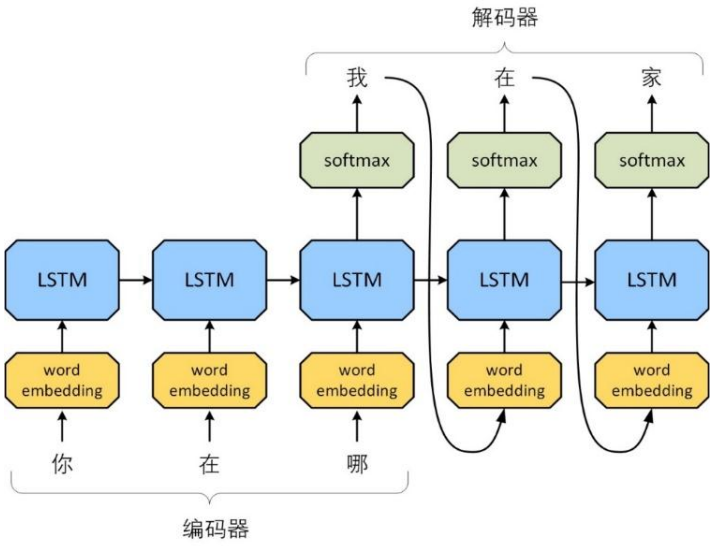


图 12.1　Seq2Seq 模型的编码器—解码器例子

对于解码生成回答的过程，Seq2Seq 模型一般采用 Beam Search 算法来完成[1]。Beam Search 算法是一种搜索算法，目的是在序列中解码出相对较优的路径。图 12.2 展示了用 Beam Search 算法解码的例子。不同于在解码时每一步只预测当前概率最高的词，对每个步骤 i，Beam Search 会在解码时记录当前概率最高的k_i个词，这k_i个词每一个后续会有可能的V个新的词，即共有$k_i \cdot V$个可能的组合。我们从V个词中取出概率最高的k_{i+1}个词组成新的组合。以此类推，直到搜索到t层。如此操作，使得它能够更好地搜索到比较好的序列。譬如在图 12.2 中，“我不告诉你”这个候选回答也有很高的概率，得分 0.162，仅排在“我在家”

1 WISEMAN S, RUSH A M, Sequence-to-sequence learning as beam-search optimization. arXiv preprint arXiv:1606.02960,2016 [C].

（得分 0.18）之后。通过 Beam Search，这样的回答也成功进入了候选回答。

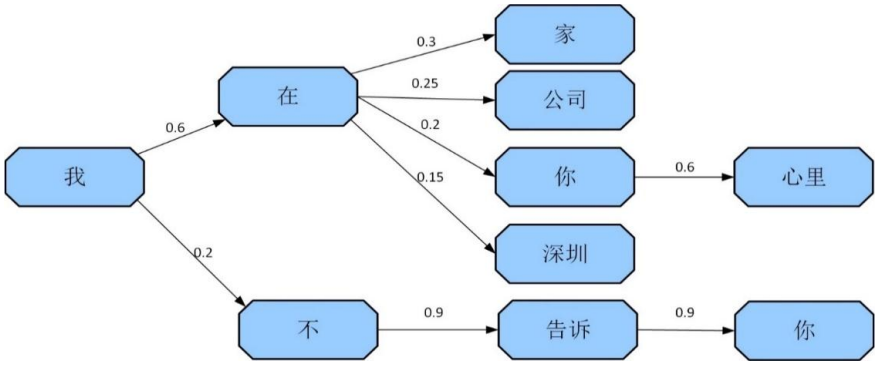


图 12.2　用 Beam Search 算法解码

Seq2Seq 模型优秀的序列生成能力也对应着巨大的计算开销。Seq2Seq 的每个节点包含一个词嵌入向量和 LSTM 结构。尤其是 LSTM 结构，其内部包含较多的运算，包括矩阵乘积、相加、合并等运算，以及σ函数和 tanh 函数的指数运算等。同时类似递归搜索，Beam Search 的搜索空间比较巨大。因为这些计算开销的存在，使得建立一个实时计算的 serving 系统很有挑战。我们初始实现的 C++版本的 Seq2Seq 解码器对每个用户查询的平均处理时间是 100 多毫秒，这样的计算开销使得其并不能直接应用在微软“必应”在线广告系统中。通过多种优化方式，我们将处理速度提升到 3.6ms，提升了 30 倍。

12.2　LSTM 优化分析

在优化中，一个重要的原则就是优化最耗时、运行时间最长的组件。譬如对于一个运行时间仅占总运行时长 10%的组件，即使优化到了极致，使得该组件的运行时间快到可以忽略了，也只能将总的运行时间降到原来的 90%。而优化一个占 90%运行时间的组件，只需要提升 11%就能达到相同的效果。从图 12.1 中可以看出，LSTM 单元对 Seq2Seq 来说就是这样一个核心组件，在 Seq2Seq 运行

的每一步都需要使用它，而且运算量最大。

优化的第一步是分析，分析什么原因导致 LSTM 计算量大、如何解决。我们在第 5 章的 5.2.3 节介绍了 LSTM 的工作过程，下面来做一个简单的回顾。图 12.3 中展示了 LSTM 的内部结构。可以看到 LSTM 主要包括三个门函数(gating function)，分别是："遗忘门"(forget gate)、"输入门"(input gate) 和 "输出门"(output gate)。每个 LSTM 单元会输出两个值，分别是记忆单元(memory cell) $\boldsymbol{C}_t$和隐藏层状态 (hidden state) $\boldsymbol{h}_t$。这两个输出会和下一个单元的新数据输入$\boldsymbol{X}_{t+1}$一起，作为下一个单元的输入。

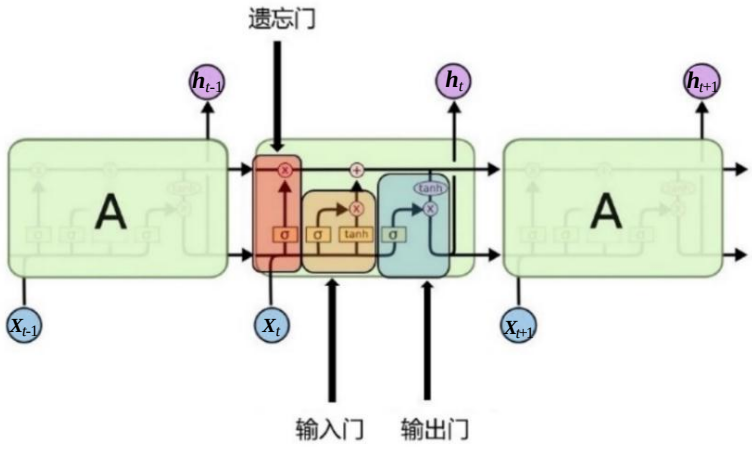


图 12.3　LSTM 内部结构

图 12.3 中红色区域为遗忘门，它选择性地保留先前状态中的信息。遗忘门通过上一个单元的隐藏层输出$\boldsymbol{h}_{t-1}$和本单元的输入$\boldsymbol{X}_t$来计算先前状态信息保留的概率$\boldsymbol{f}_t$，$\boldsymbol{f}_t$为(0,1)之间的数值，越接近 0 越容易遗忘，越接近 1 则保留越多。$\boldsymbol{f}_t$会用来乘以$\boldsymbol{C}_{t-1}$，如公式(4)所示，表示保留上一个单元信息的比例。譬如，$\boldsymbol{f}_t$为 0.4，表示本单元会保留上一个单元 40%的信息。

$$f_t = \sigma\left(\boldsymbol{W}_f[\boldsymbol{h}_{t-1}, \boldsymbol{X}_t] + b_f\right) \tag{1}$$

遗忘门的现实意义在于，考虑在语言模型中，基于已经看到的信息预测下一个词。先前状态$\boldsymbol{h}_{t-1}$中的一部分信息可能包含当前主语的类别。当我们看到新的主语时，希望忘记旧的主语。例如，“他今天有事，所以我……”当处理到“我”的时候选择性地忘记前面的“他”，或者说减小“他”这个词对后续词的作用。而在另一个例子“我吃……”中，我们希望记住“我”的信息，因为“我”与“熊猫”“松鼠”不同。

图 12.3 中橙色区域为输入门，其作用是将新的信息选择性地记录到记忆单元中。输入门将会通过当前单元的输入$\boldsymbol{X}_t$来计算本单元的状态$\widetilde{\boldsymbol{C}_t}$（公式 2）及比例$i_t$（公式 3）。它们的乘积$i_t\widetilde{\boldsymbol{C}_t}$将会加上遗忘门的输出$f_t\boldsymbol{C}_{t-1}$一起，得到本单元的最终状态$\boldsymbol{C}_t$，如下所示。

$$\widetilde{\boldsymbol{C}_t} = \tanh(\boldsymbol{W}_C[\boldsymbol{h}_{t-1}, \boldsymbol{X}_t] + b_C) \tag{2}$$

$$i_t = \sigma(\boldsymbol{W}_i[\boldsymbol{h}_{t-1}, \boldsymbol{X}_t] + b_i) \tag{3}$$

$$\boldsymbol{C}_t = \boldsymbol{f}_t\boldsymbol{C}_{t-1} + \boldsymbol{i}_t\widetilde{\boldsymbol{C}_t} \tag{4}$$

输入门的现实意义很容易理解，当我们引入新词时，也引入了新的信息，譬如“我吃……”。当看到“吃”的时候，我们知道后续应该跟着某种食物，譬如“饭”“水果”。引入“吃”这个信息能够更加确定“我”做事的范围，而不可能是“我爱你”“我在家”等这样的情况。

图 12.3 中蓝色区域为输出门，它决定下一个隐藏状态是什么。为此，它整合了先前的隐藏状态$\boldsymbol{h}_{t-1}$，当前输入$\boldsymbol{X}_t$，以及本单元状态$\boldsymbol{C}_t$。它将tanh函数输出与σ函数输出相乘，以决定隐藏状态应携带的信息。譬如，在“我吃……”这个例子中。之前出现的词“我”和当前的“吃”共同决定了之后出现的词的范围。

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o[\boldsymbol{h}_{t-1}, \boldsymbol{X}_t] + b_o) \tag{5}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \tanh(\boldsymbol{C}_t) \tag{6}$$

LSTM 中的每个函数都有其对应的实际解释。当我们认真去观察这些函数运算时，会发现其中包含了大量的指数运算（譬如，σ函数和tanh函数）和大量的矩阵运算（譬如，$\boldsymbol{W}_f[\boldsymbol{h}_{t-1}, \boldsymbol{X}_t]$等）。所以，当我们试图优化 LSTM 运行速度时，最需要优化的就是这些数学运算。

与 LSTM 类似的是，词嵌入向量也是一个矩阵运算的过程，其在 8.2.1 节已有阐述，本章不再赘述；其优化方法亦可参考本章 LSTM 的优化方法。

12.2.1 优化一：指数运算的近似展开

我们来看一下如何加快σ函数和tanh函数的计算速度。

在计算机中，对于指数、对数、开方之类的运算并没有精确的结果，而是采用近似的方式逼近足够的精度，常用的方法包含牛顿法、泰勒展开等。计算机计算复杂函数的关键是要了解实际需要多少精度，然后选择合适的近似方法来近似计算。在这里，我们使用连分数来快速近似计算σ函数和tanh函数[1]。本方法只需要少量计算就能使误差小于0.011%。

连分数常用于无理数的逼近，它能够快速逼近真实值。我们可以用连分数来表示$\tanh(x)$，其连分数形式如下：

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{x^2}{5 + \cfrac{x^2}{7 + \cfrac{x^2}{9 + \cfrac{x^2}{11 + \cfrac{x^2}{13 \cdots}}}}}}} \tag{7}$$

1 TEMURTAS F, GULBAG A, YUMUSAK N, A study on neural networks using taylor series expansion of sigmoid activation function. InInternational Conference on Computational Science and Its Applications 2004 [C] . Springer, Berlin, Heidelberg, 389-397.

连分数具有很好的特性，通过归纳法可知，在$x>0$ 时，其奇数层展开均大于$\tanh(x)$的真实值，并逐渐缩小。其原因是，以$\tanh(x)$的第一层展开为例，$\frac{x}{1}$大于真实值，因为分母 1 比$1+\frac{x^2}{3+\cdots}$小，所以整个分式大于$\tanh(x)$的真实值；偶数层展开均小于$\tanh(x)$的真实值，并逐渐增大。其原因是，以$\tanh(x)$的第 2 层展开为例，$\frac{x}{1+\frac{x^2}{3}}$，其分母为$1+\frac{x^2}{3}$，它大于$1+\frac{x^2}{3+\cdots}$，所以整个分式小于$\tanh(x)$的真实值。

记$\tanh_j(x)$表示$\tanh(x)$的第j层展开，j和n是大于 1 的整数，于是有：

$$\tanh_2(x)<\cdots<\tanh_{2n}(x)<\tanh(x)<\tanh_{2n-1}(x)<\cdots<\tanh_1(x) \tag{8}$$

我们将证明，将$\tanh(x)$展开到第 7 层，其与真实值的误差将小于 0.011%。

记$\tanh_7(x)$和$\tanh_8(x)$分别是$\tanh(x)$第 7 层展开和第 8 层展开的值，我们知道其有$\tanh_8(x)<\tanh(x)<\tanh_7(x)$，即：

$$\frac{x}{1+\cfrac{x^2}{3+\cfrac{x^2}{5+\cfrac{x^2}{7+\cfrac{x^2}{9+\cfrac{x^2}{11+\cfrac{x^2}{13+\cfrac{x^2}{15}}}}}}}}<\tanh(x)\frac{x}{1+\cfrac{x^2}{3+\cfrac{x^2}{5+\cfrac{x^2}{7+\cfrac{x^2}{9+\cfrac{x^2}{11+\cfrac{x^2}{13}}}}}}} \tag{9}$$

我们对$\tanh_7(x)$ 和$\tanh_8(x)$展开为普通分数，有：

$$\begin{aligned}\tanh_7(x)&=\frac{x}{1+\cfrac{x^2}{3+\cfrac{x^2}{5+\cfrac{x^2}{7+\cfrac{x^2}{9+\cfrac{x^2}{11+\cfrac{x^2}{13}}}}}}}\\&=\frac{x\cdot(135135+17325x^2+378x^4+x^6)}{135135+62370x^2+3150x^4+28x^6}\end{aligned} \tag{10}$$

$$\tanh_8(x)=\cfrac{x}{1+\cfrac{x^2}{3+\cfrac{x^2}{5+\cfrac{x^2}{7+\cfrac{x^2}{9+\cfrac{x^2}{11+\cfrac{x^2}{13+\cfrac{x^2}{15}}}}}}}}=\frac{x\cdot(2027025+270270x^2+6930x^4+36x^6)}{2027025+945945x^2+51975x^4+630x^6+x^8} \tag{11}$$

所以，$\tanh(x)$与$\tanh_7(x)$的差异是：

$$\begin{aligned}&|\tanh(x)-\tanh_7(x)|<\tanh_7(x)-\tanh_8(x)\\&=\frac{x^{15}}{(2027025+945945x^2+51975x^4+630x^6+x^8)\cdot(135135+62370x^22+3150xx^4+28xx^6)}\\&=\frac{x^{15}}{273922023375+254255826825x^2+72407360025x^4+6363299250x^6+229635945x^8+3502170x^{10}+20790x^{12}+x^{14}}\end{aligned} \tag{12}$$

由此，我们可以估算$\tanh_7(x)$的精度：首先，当$x\geqslant 5$时，$\tanh(x)>0.999909204$，它与 1 的精度误差小于 0.0091%，可以直接返回 1。同样，当$x\leqslant -5$时，可以直接返回-1。

当x在[5，−5]范围中，我们可以用公式（12）来估算$\tanh(x)$与$\tanh_7(x)$差异的上限。当x=5 时：

$$\begin{aligned}&|\tanh(x)-\tanh_7(x)|<|\tanh_7(x)-\tanh_8(x)|\leqslant\tanh_7(5)-\tanh_8(5)\\&=\frac{30517578125}{280460470444000}<0.011\%\end{aligned} \tag{13}$$

考虑到$\tanh_7(x)-\tanh_8(x)$是一个随着x的增大而增大的函数，所以当$|x|<5$时，$|\tanh(x)-\tanh_7(x)|<|\tanh_7(x)-\tanh_8(x)|<0.011\%$，于是我们可以用$\tanh_7(x)$近似表示$\tanh(x)$：

$$\begin{aligned}\tanh(x)\approx\tanh_7(x)&=\frac{x^7+378x^5+17325x^3+135135x}{28x^6+3150x^4+62370x^2+135135}\\&=\frac{\left(\left((x^2+378)x^2+17325\right)x^2+135135\right)x}{\left((28x^2+3150)x^2+62370\right)x^2+135135}\end{aligned} \tag{14}$$

我们将公式（14）改写成代码，对$\tanh(x)$的近似计算代码实现如下：

```
inline float fast_tanh_contfrac7(float x) {
    if (x >= 5.0f) return 1.0f;
    if (x <= -5.0f) return -1.0f;
    float x2 = x * x;
    float a = x * (135135.0f + x2 * (17325.0f + x2 * (378.0f + x2)));
    float b = 135135.0f + x2 * (62370.0f + x2 * (3150.0f + x2 * 28.0f));
    return a / b;
}
```

在代码中，我们只用了 7 个乘法操作、6 个加法操作和 1 个除法操作就完成了对$\tanh(x)$的求解。有了对$\tanh(x)$函数的计算之后，可以很容易地计算$\sigma(x)$函数。我们知道$\sigma(x)$函数和$\tanh(x)$函数之间可以相互转换，即：

$$\sigma(x)=\frac{e^{x}}{e^{x}+1}=\frac{e^{\frac{x}{2}}}{e^{\frac{x}{2}}+e^{-\frac{x}{2}}}=\frac{\tanh\left(\frac{x}{2}\right)+1}{2} \tag{15}$$

于是，可以使用与计算$\tanh(x)$相似的方法来计算$\sigma(x)$。代码实现如下：

```
inline float fast_sigmoid_contfrac7(float x) {
    if (x >= 10.0f) return 1.0f;
    if (x <= -10.0f) return -0.0f;
    x = 0.5f * x;
    float x2 = x * x;
    float a = x * (135135.0f + x2 * (17325.0f + x2 * (378.0f + x2)));
    float b = 135135.0f + x2 * (62370.0f + x2 * (3150.0f + x2 * 28.0f));
    return 0.5f * (a / b + 1);
}
```

为了保持足够的精度，我们将近似计算x在$(-10, 10)$之间时$\sigma(x)$的值；而在$x\geqslant 10$或$x\leqslant -10$时，直接返回 1 或−1。由公式（15）的$\sigma(x)$函数与$\tanh(x)$函数的转换可知，这个计算范围与计算$\tanh(x)$的范围一致。

同样，我们也可以计算函数 fast_sigmoid_contfrac7(x)的精度。

当$x\geqslant 10$ 时，$\sigma(x)>0.999954602$。它与 1 的精度误差小于 0.00454%，类似地，当$x\leqslant -10$时，$\sigma(x)<0.0000453978687$，精度误差小于 0.00454%。

当x为$(-10, 10)$时，我们可以通过公式（15）转换成$\tanh(x)$函数来估算其精度误差：

$$|\sigma(x)-\sigma_7(x)|=\left|\frac{\tanh\left(\frac{x}{2}\right)+1}{2}-\frac{\tanh_7\left(\frac{x}{2}\right)+1}{2}\right|$$
$$=\frac{\left|\tanh\left(\frac{x}{2}\right)-\tanh_7\left(\frac{x}{2}\right)\right|}{2}<\frac{\left|\tanh\left(\frac{10}{2}\right)-\tanh_7\left(\frac{10}{2}\right)\right|}{2}<\frac{0.011\%}{2} \quad (16)$$
$$=0.0055\%$$

在本节中，我们阐述了如何使用一些数学变换，对复杂的指数运算展开成少量步骤的多项式运算，并达到比较高的精度。事实上，展开成多项式运算不仅仅能加快运算速度，在 12.2.2 节中，我们将阐述如何将其与矢量化技术结合，进行多个数据的并行运算，进一步提高运算速度。

12.2.2 优化二：矩阵运算的执行速度优化

本节我们优化计算机执行矩阵运算的速度。

在实践中，人们对优化计算机执行矩阵运算的速度已经做了相当长时间的研究和努力。这里值得一提的是 BLAS 库，其对深度学习运算速度的优化很有帮助。BLAS 的全称是“基础线性代数子程序库”（basic linear algebra subprograms），首次推出于 1979 年，提供了一些底层的通用线性代数运算的函数实现，如向量的相加、数乘、点积和矩阵相乘等。BLAS 的实现根据硬件平台的不同而不同，常常利用了特定处理器的硬件特点进行加速计算。常见的 BLAS 加速库有英特尔的 MKL、AMD 的 LibM、ATLAS 和 OpenBLAS。2017 年 3 月，英特尔开源了专为 DNN 使用的 MKL-DNN 开源库。

本章对 Seq2Seq 的加速基于英特尔的 CPU 机器，我们将以英特尔 MKL（Intel Math Kernel Library，英特尔数学核心函数库）为例阐述如何对 Seq2Seq 进行加速。

对矩阵运算来说，在代码实现中，单纯用 for 循环实现效率太低，使用 MKL 能够大大加快矩阵运算的速度。举一个例子，有两个二维矩阵 matrix_a 和

matrix_b，其中 matrix_a 的维度为 row×inner，matrix_b 的维度是 inner×col，它们乘法的结果为 matrix_c（维度是 row×col）。

假如我们简单地使用 for 循环，其代码为三重循环：

```
   inline void MatrixMult(float *matrix_a, float *matrix_b, float *matrix_c, int row,
int col, int inner) {
       for (int i = 0; i < row; i++) {
           for (int j = 0; j < col; j++) {
               matrix_c[i * col + j] = 0;
               for (int k = 0; k < inner; k++) {
                   matrix_c[i * col + j] += matrix_a[i * inner + k] * matrix_b[k * col
+ j];
               }
           }
       }
   }
```

若用 MKL，我们可以使用 cblas_sgemm()函数重写矩阵运算：

```
   inline void MatrixMultMkl(float *matrix_a, float *matrix_b, float *matrix_c, int
row, int col, int inner) {
       float alpha = 1.0f;
       float beta = 0.0f;
       cblas_sgemm(
           CblasRowMajor,
           CblasNoTrans,
           CblasNoTrans,
           row, col, inner, alpha,
           matrix_a, inner,
           matrix_b, col, beta,
           matrix_c, col
       );
   }
```

经过对比，MatrixMult()函数和 MatrixMultMkl()函数的速度相差近 100 倍。第二种实现我们基本直接调用 cblas_sgemm()函数，其接口如下：

```
   void cblas_sgemm(const enum CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA,
const enum CBLAS_TRANSPOSE TransB, const int m, const int n, const int k, const float
alpha, const float  *A, const int lda, const float  *B, const int ldb, const float beta,
float  *C, const int ldc)
```

这个函数实现的操作是$\boldsymbol{C}$ = alpha×op($\boldsymbol{A}$)×op($\boldsymbol{B}$) + beta×$\boldsymbol{C}$。其中，

op($\boldsymbol{X}$)是矩阵$\boldsymbol{X}$三个操作的一种，其中$\boldsymbol{X}$为$\boldsymbol{A}$或者$\boldsymbol{B}$：

若 Trans$\boldsymbol{X}$为 CblasNoTrans，则 op($\boldsymbol{X}$) = $\boldsymbol{X}$（$\boldsymbol{X}$本身）。

若 Trans$\boldsymbol{X}$为 CblasTrans，则 op($\boldsymbol{X}$) = $\boldsymbol{X}^{\mathrm{T}}$（$\boldsymbol{X}$的转置）。

若 Trans$\boldsymbol{X}$为 CblasConjTrans，则 op($\boldsymbol{X}$) =$\boldsymbol{X}^{\mathrm{H}}$（$\boldsymbol{X}$的共轭转置）。

其中，alpha 和 beta 是标量；

$\boldsymbol{A}$、$\boldsymbol{B}$、$\boldsymbol{C}$是三个矩阵，其中 op($\boldsymbol{A}$)是一个$m\times k$的矩阵，op($\boldsymbol{B}$)是一个$k\times n$的矩阵，$\boldsymbol{C}$是一个$m\times n$的矩阵。

【知识点讲解】为什么 MKL 很高效

那么，为什么 MKL 在 sgemm（单精度矩阵乘法）和 dgemm（双精度矩阵乘法）中能高效运行呢？简单地说：因为它有效地利用了数据缓存和矢量化。

知识点一：MKL 与数据缓存

我们知道计算机的存储结构为寄存器、高速缓存、内存、硬盘多级结构。在多级层次结构中，由上向下，其容量逐渐增大，成本逐次减少，速度则逐级降低。这个设计可以说是颇为完美的设计，兼顾了速度与成本。可是也并非十全十美，为了能让计算机达到理想的执行速度，我们需要尽量将待处理的数据提前从内存预取到高速缓存中。

在矩阵很大时，整个矩阵不能放在缓存中，程序必须从主存中获取所需的元素，从而损失大量时间。在实际使用时，我们希望计算机是“聪明”的，它能够有效地把即将要运行的内容导入到缓存中。如果要取的内容并不在内存中连续出现，计算机将会出现多次缓存失效的情况。譬如，在矩阵乘法中，我们知道内存中二维数组是以行为单位连续存储的，逐列访问将会每次越过多个地址来寻取元素。在我们实现的函数 MatrixMult()中，matrix_b[k*col + j] 将每次跳 col 个元素：当前元素是 matrix_b[k*col + j]，下一个元素将是 matrix_b[(k+1)*col + j]。根据 CPU 缓存的替换策略，这会导致出现大量的缓存失效。为此，我们可以做一个简单优化：调换一下循环的执行顺序，当前元素 matrix_b[k*col + j]，取下

一个元素为 matrix_b[k*col + j+1]。这样，对矩阵的访问将位于连续的内存地址中，可以同时载入缓存。CPU 对缓存中的数据访问比内存中快很多，这将大大加快矩阵运算的速度。

除让矩阵运算在内存中连续访问外，为了避免矩阵很多时候常出现的缓存数据失效，将矩阵分块也是一个常用的优化，譬如对矩阵$\boldsymbol{A}$分成 4 块，即$\boldsymbol{A}=\begin{pmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{pmatrix}$，或者更多块。矩阵$\boldsymbol{B}$也可以做如此切分：$\boldsymbol{B}=\begin{pmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{pmatrix}$。在代数中，我们知道，矩阵$\boldsymbol{AB}$的乘法结果是对应的分块矩阵的乘积和，即为$\boldsymbol{AB}=\begin{pmatrix}A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22}\\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22}\end{pmatrix}$。如果切分合理，每一次运算时$\boldsymbol{A}$和$\boldsymbol{B}$的分块矩阵都能加载到缓存中。这样，对于计算机来说，其所需的数据都在缓存中，有效地减少了缓存失效的状况，减少了等待数据的时间，大大加快矩阵的运算速度。

以上是关于高性能计算库 BLAS 对大型矩阵运算基于缓存优化的两个例子，在实际应用 BLAS 实现时，其用到的技术会更复杂，也更为高效。

知识点二：MKL 与矢量化

矢量化又称单指令流多数据流（Single Instruction Multiple Data，SIMD）。它是一种采用一个控制器来控制多个处理器，同时对一组数据（“数据向量”）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。在微处理器中，单指令流多数据流技术使用一个控制器控制多个平行的处理微元，例如英特尔的 AVX 或 SSE，以及 AMD 的 3D Now!指令集。

SIMD 技术的关键是在一条单独的指令中同时执行多个数据的运算操作，以增加处理器的吞吐量。为了解 SIMD 在性能上的优势，我们以乘法指令为例，针对多对数据同时执行乘法运算的任务，对比使用 SIMD 技术与否的区别。在未使用 SIMD 技术时，乘法指令译码后，执行部件每次访问主存，取得两个操作数，然后进行乘法运算，再取新的两个操作数进行乘法运算，如此循环；而使用了 SIMD 技术，在指令译码后，几个执行部件同时访问主存，一次性获得所有操作

数进行一次运算。这一特点使得 SIMD 技术特别适合于机器学习这样的数据密集型运算。

合理使用 SIMD 能够大大加快程序的执行速度。以英特尔的 AVX 为例，它的每个指令能够处理 256 比特的数据。譬如，我们有两组数据向量$\boldsymbol{a}$,$\boldsymbol{b}$，每组数据向量包含 8 个 float 类型的数据，每个数据占 32 比特：

$$\boldsymbol{a} = [a_0, a_1, a_2, \cdots, a_7]$$
$$\boldsymbol{b} = [b_0, b_1, b_2, \cdots, b_7]$$

我们希望它们执行乘法运算，得到数据向量$\boldsymbol{c}$：

$$\boldsymbol{c} = [a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, \cdots, a_7 \cdot b_7]$$

如果我们对每个数据分别单独执行乘法运算，则共需要 8 条乘法指令：

```
for (int i = 0; i < 8; i++) {
   c[i] = a[i] * b[i];
}
```

但是，如果使用 AVX，即这些数据的运算可以由一条指令完成：

```
#include <immintrin.h>
_ _m256 c = _mm256_mul_ps(a, b)
```

如此，大大加快了运算速度。SIMD 要求对多个数据执行相同的操作，这样的特性使得它很适合处理矩阵运算。

在 Seq2Seq 模型的 Beam Search 算法中，SIMD 可用于提高运算的速度。我们以$\tanh(x)$函数为例。在 12.2.1 节“指数运算的近似展开”中，我们已经阐述了如何近似$\tanh(x)$函数并达到足够的精度。下面将使用 SIMD 改写 fast_tanh_contfrac7(x)函数，使其能够支持多条数据在单指令里的同时执行。

```
#define __Init_Contfrac7_Avx_Bounds__ \
__m256 ub = _mm256_set1_ps(5.0f); \
__m256 lb = _mm256_set1_ps(-5.0f); \
__m256 xsq, xnum

#define __Init_Contfrac7_Avx_Constants__ \
```

```
__m256 c28 = _mm256_set1_ps(28.0f); \
__m256 c378 = _mm256_set1_ps(378.0f); \
__m256 c3150 = _mm256_set1_ps(3150.0f); \
__m256 c17325 = _mm256_set1_ps(17325.0f); \
__m256 c62370 = _mm256_set1_ps(62370.0f); \
__m256 c135135 = _mm256_set1_ps(135135.0f);

#define __Fast_Tanh_Contfrac7_Avx__(x1, x3) \
x1 = _mm256_min_ps(x1, ub); \
x1 = _mm256_max_ps(x1, lb); \
xsq = _mm256_mul_ps(x1, x1); \
xnum = _mm256_add_ps(xsq, c378); \
xnum = _mm256_mul_ps(xnum, xsq); \
xnum = _mm256_add_ps(xnum, c17325); \
xnum = _mm256_mul_ps(xnum, xsq); \
xnum = _mm256_add_ps(xnum, c135135); \
xnum = _mm256_mul_ps(x1, xnum); \
x3 = _mm256_mul_ps(xsq, c28); \
x3 = _mm256_add_ps(x3, c3150); \
x3 = _mm256_mul_ps(x3, xsq); \
x3 = _mm256_add_ps(x3, c62370); \
x3 = _mm256_mul_ps(x3, xsq); \
x3 = _mm256_add_ps(x3, c135135); \
x3 = _mm256_div_ps(xnum, x3);
```

下面来解释一下__Fast_Tanh_Contfrac7_Avx__宏函数，它有两个参数x1和x3,每个分别是256位的矢量。每个矢量包含了8个float类型的数据。其中x1包含$\tanh(x)$函数的输入数据，$\tanh(x)$函数的计算结果会保存在x3中。

比较有意思的是__Fast_Tanh_Contfrac7_Avx__函数的第1行与第2行，在这里，我们并没有采用如fast_tanh_contfrac7(x)函数中使用if语句进行判断直接返回1或-1的方式，而是通过最小化（min）和最大化（max）操作，将输入数据限制在[-5, 5]之间。其原因是，在多个数据输入的情况下，很可能出现有个别输入数据不在[-5, 5]之间，其他数据在[-5, 5]之间的情况。这时候并不能直接返回一个全为1或者-1的矢量。

我们在12.2.1节中分析过，当$x \geqslant 5$或$x \leqslant -5$时，$\tanh(x)$与1或-1的误差不超过0.0091%，而$\tanh(x)$是单调递增函数，当$x > 5$时，有$\tanh(5) < \tanh(x) < 1$，即使我们用$\tanh(5)$代替$\tanh(x)$，其与真实值的差异也不会超过0.0091%.

由公式（10）可知，在近似计算$\tanh(x)$时，需要乘以一些常量，包括 378、17325、28、3150 等。对于这些常量，我们可以通过_mm256_set1_ps(c)函数进行初始化，生成一个形如<c,c,c,c,c,c,c,c>的矢量。定义了这些常量之后，用 SIMD 实现$\tanh(x)$就变得很容易，只需要将加减乘除方法分别用 SIMD 函数对应起来即可：加方法(_mm256_add_ps)；减方法(_mm256_sub_ps)；乘方法(_mm256_mul_ps)；除方法(_mm256_div_ps)。类似的方法同样可以用于实现$\sigma(x)$函数。

使用 SIMD 的一个小建议是，对数据做好对齐，即保证需要处理的向量起始地址是 SIMD 类型宽度的整数倍。譬如，SSE 需要 16 字节对齐，AVX 需要 32 节对齐，AVX-512 需要 64 字节对齐。为了方便起见，我们也可以统一采用 64 比特对齐，如对 float 类型向量 a 对齐，如下：

```
float *a = (float *)mkl_malloc(size_a * sizeof(float), 64);
```

或者我们也可以采用编译制导语句交于编译器来帮助完成数据对齐：

```
#pragma simd
#pragma vector aligned
```

12.2.3 优化三：多线程并行处理

当前的处理器一般具有多个核，如果能够合理地利用多核进行多线程的并行处理，就能取得几倍甚至十几倍的加速。譬如，在理想情况下，如果两个多线程之间没有数据依赖，不会相互阻塞，那就能提高一倍的处理速度。

目前，许多库函数都支持多线程，譬如 boost::thread 库或者 C++11 以后的 std::thread 库，以及操作系统相关的线程 API，如 Linux 的 pthread 库等。这里我们使用了另一种支持多线程 API：OpenMP（Open Multi-Processing）。相比其他多线程库，其优势在于，它能够为编写多线程程序提供一种简单的方法，而无须程序开发人员进行复杂的线程创建、同步、负载均衡和销毁工作。

【知识点讲解】OpenMP 简介

OpenMP 由一系列#pragma 指令组成，提供了对于并行描述的高层抽象，降低了并行编程的难度和复杂度：开发人员可以通过在源代码中加入专用的#pragma 指令来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。当选择忽略这些#pragma 指令，或者编译器不支持 OpenMP 时，程序又可退化为通常的程序，代码仍然可以正常运作，只是不能利用多线程来加速程序执行而已。

下面我们来看一个使用 OpenMP 的例子：

```
#include <omp.h>
#include <iostream>
using namespace std;

int main() {
#pragma omp parallel for
    for (int i = 0; i < 10; ++i)
    {
        cout << i;
    }
    cout << endl;
    return 0;
}
```

上述代码中，我们通过#pragma omp 预处理指示符指定编译器采用 OpenMP，并通过#pragma omp parallel for 来指定下方的 for 循环采用多线程执行，此时编译器会根据 CPU 的个数来创建线程数。譬如，对于双核系统，编译器会默认创建两个线程执行并行区域的代码。

在深度学习的任务中，很多运算都可以拆分成多个“没有数据依赖”的运算，常见的有多个数据样本的并行处理和矩阵运算等。

1. 多线程处理多个数据样本

对多个数据样本的多线程并行化处理非常直观，这些样本相互独立，互不干扰。譬如用 Seq2Seq 模型同时处理不同的用户查询时，这些查询是相互独立的，其结果不会也不应该会被其他的查询所影响。其伪程序如下：

```
#pragma omp parallel for num_threads(nt)
for (int i = 0; i < batch_size; i++) {
        获取第 i 个查询数据
        对第 i 个查询数据进行处理
    }
```

与样例程序类似，我们只需要在原来的程序中加入一行代码#pragma omp parallel for num_threads(nt)，其中 nt 表示线程个数，多个数据样本的处理就可以通过多线程并行完成。

2. 多线程处理矩阵运算

回到我们之前讨论的分块矩阵，矩阵分块除能够避免缓存失效外，更出色的性质是能够利用多线程进行并行处理。考虑 12.2.2 节的分块矩阵乘法的例子，对于矩阵 $\boldsymbol{A}=\begin{pmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{pmatrix}$，矩阵 $\boldsymbol{B}=\begin{pmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{pmatrix}$。其乘法结果即为：$\boldsymbol{C}=\boldsymbol{AB}=\begin{pmatrix}A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22}\\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22}\end{pmatrix}=\begin{pmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{pmatrix}$

观察结果，我们能发现，A_{11}和A_{12}只参与了对C_{11}和C_{12}的运算，A_{21}和A_{22}只参与了对C_{21}和C_{22}的运算；B_{11}和B_{21}只参与了对C_{11}和C_{21}的运算，B_{12}和B_{22}只参与到了对C_{12}和C_{22}的运算。只要划分好数据，C_{11}、C_{12}、C_{21}、C_{22}的计算就可以并行完成，这个性质使得我们可以方便地采用多线程来帮助矩阵运算。即只要对 $\boldsymbol{A}$ 矩阵按行划分子矩阵，或者对 $\boldsymbol{B}$ 矩阵按列划分子矩阵，其运算将不互相干扰：

$$\boldsymbol{A}=\begin{pmatrix}\boldsymbol{A}_1\\ \cdots\\ \boldsymbol{A}_{\text{row}}\end{pmatrix},\boldsymbol{C}=\boldsymbol{AB}=\begin{pmatrix}\boldsymbol{A}_1\boldsymbol{B}\\ \cdots\\ \boldsymbol{A}_{\text{row}}\boldsymbol{B}\end{pmatrix} \tag{17}$$

或者，

$$\boldsymbol{B}=(\boldsymbol{B}_1 \quad \cdots \quad \boldsymbol{B}_{\text{col}}),\boldsymbol{C}=\boldsymbol{AB}=(\boldsymbol{AB}_1 \quad \cdots \quad \boldsymbol{AB}_{\text{col}}) \tag{18}$$

对公式（17）的函数实现如下：

```
inline void MatrixMultOmp(float *matrix_a, float *matrix_b, float *matrix_c, int
row, int col, int inner, int nt) {
    int long_len = (row + nt - 1) / nt;
    int gap = nt * long_len - row;
    float alpha = 1.0f, beta = 0.0f;
```

```
    #pragma omp parallel for num_threads(nt)
        for (int i = 0; i < nt; i++) {
            int len = (i >= gap) ? long_len : long_len - 1;
            int diff = (i >= gap) ? gap : i;
            float * matrix_a_i = matrix_a + (long_len *i - diff) * inner;
            float * matrix_c_i = matrix_c + (long_len *i - diff) * inner;
            cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                len, col, inner, alpha, matrix_a_i, inner, matrix_b, col, beta,
matrix_c_i, col);
        }
    }
```

我们来解释一下 MatrixMultOmp()函数，这个函数比 12.2.2 节的 MatrixMultMkl()函数多了一个参数 nt，表示线程个数。在具体实现时，考虑到总行数 row 不一定能被 nt 均分，我们对行的划分原则是各个子矩阵的行数尽可能接近，它们的区别在一行之内，最小行数$\frac{\mathrm{row}}{\mathrm{nt}}$，最大行数$\frac{\mathrm{row}}{\mathrm{nt}}+1$。

在对矩阵进行分割之后，我们可以计算各个子矩阵的起始位置，即原始矩阵指针 matrix_a 加上偏移量(long_len *i - diff) * inner。然后使用 cblas_sgemm 函数进行运算。

根据实际情况，我们也可以对公式（18）进行类似的实现。

12.3 优化应用实例：RapidScorer 算法对 GBDT 的加速

在本节中，我们将本章使用的优化技术，尤其是 SIMD 部分的技术应用于另一个工作——GBDT 的加速中。

GBDT，即梯度提升决策树（gradient boosting decision tree），由多棵决策树组成，所有树的预测结果累加起来作为最终答案。GBDT 是一个非常经典的模型，对一个经典的模型进行速度上的优化颇为困难。在前人的基础上，我们提出了一个名为 RapidScorer[1]的新算法，发表于 KDD 2018。RapidScorer 算法相

1 YE T, ZHOU H C, ZOU W Y, et al, Rapidscorer: fast tree ensemble evaluation by maximizing compactness in data level parallelization. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining 2018[C], ACM, 941-950.

比于其他算法有明显的速度提升：V-QuickScorer[1]，提升了 1.3~3.5 倍；相比于 QuickScorer[2]，提升了 2.1~25.0 倍；相比于 VPred[3]，提升了 2.3~18.3 倍；相比于 XGBoost[4]，提升了 2.6~42.5 倍。

在本实例中，我们的主要优化包含两个方面：①设计了全新的数据存储结构，节省了内存消耗并加快运算；②使用 SIMD 指令实现了 RapidScorer 算法的每一步计算。

12.3.1 背景介绍

在介绍 RapidScorer 算法之前，我们先简单介绍一下 RapidScorer 的前身——QuickScorer，发表于 SIGIR 2015。QuickScorer 算法的最大优点是，它提供了一种顺序执行、顺序访问内存数据的方法，最大化地利用了缓存。我们在 12.2.2 节的“MKL 与数据缓存”中谈到，数据和指令的顺序访问能够大大地加快程序的处理速度。我们先来看一下经典的树遍历方式，经典的树遍历方式是从根节点出发，在每个节点进行判断，根据比较结果来判定访问左子树还是右子树。这个遍历方式直观简洁，但是对计算机来说，这种方式存在控制依赖，在每一个处理

1 LUCCHESE C, NARDINI F M, ORLANDO S, et al, Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles. Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval 2016[C]. ACM, 833-836.

2 LUCCHESE C, NARDINI F M, ORLANDO S, et al, Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval 2015[C]. ACM, 73-82.

3 LUCCHESE C, NARDINI F M, ORLANDO S, et al, Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval 2015[C]. ACM, 73-82.

4 CHEN T Q, GUESTRIN C, Xgboost: A scalable tree boosting system. InProceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining 2016[C]. ACM, 785-794.

的节点都需要经过判断才能确定下一步需要处理的节点，这使得计算机指令流水线常常断流，不能最大限度地发挥流水线的效率。

为了充分发挥计算机指令流水线的效率，QuickScorer 提出了一个非常巧妙的树节点表示方法，叫作 nodemask，它将每个内部节点表示成一串 01 字符，如图 12.4 所示，n_0的 nodemask 是 0001111，n_2的 nodemask 是 1110011。nodemask 将树的遍历变成了一系列连续的位运算操作，最后运算结果的第一个 1 出现的位置即为该输入数据落到的叶节点位置。计算过程如图 12.4 右上方所示。

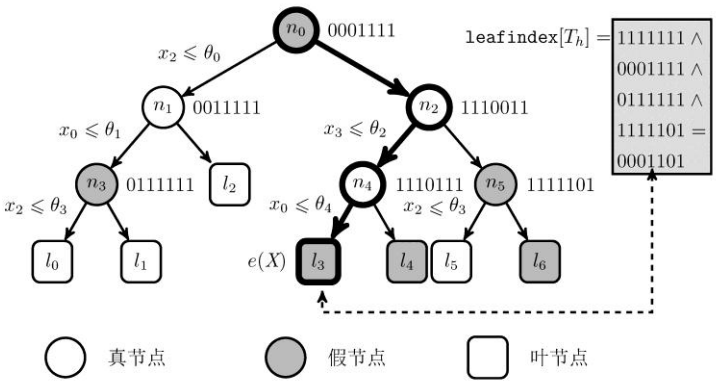


图 12.4　QuickScorer 节点表示方式

针对输入数据，QuickScorer 将每个内部节点分成了真节点（true node）和假节点（false node）。真假节点的定义为输入数据在该节点的布尔测试是否为真。对于给定一个输入数据 X，若 X 在节点n_i判断结果为真，则n_i为真节点，下一步将访问左子树；若判断结果为假，则n_i为假节点，下一步将访问右子树。在图 12.4 中，若输入数据 X 满足$\{\theta_0 < x_2 \leqslant \theta_3, x_3 \leqslant \theta_2, x_0 \leqslant \theta_1, x_0 \leqslant \theta_4\}$，于是有$n_1$、$n_2$、$n_4$为真节点；$n_0$、$n_3$、$n_5$为假节点。QuickScorer 用一个节点掩码(nodemask)来表示每一个节点n_i，其定义是，nodemask 的长度等于叶节点个数Λ，nodemask 的每一个比特与一个叶节点相对应：若叶节点l_j位于该内部节点n_i的左子树上，则为 0；若l_j不在n_i的左子树上，则为 1。怎么理解 nodemask 呢？我们可以将 nodemask

看成一个屏蔽码，用来选择是否屏蔽左子树的叶节点，如果该节点为假节点，其左子树将不会被访问，左子树的所有叶节点也不会被访问。有了假节点和 nodemask 的定义之后，QuickScorer 定义每棵树T_h对每个输入数据有一个初始全为 1 的向量$\mathbf{leafindex}[T_h]$，通过$\mathbf{leafindex}[T_h]$和所有假节点的 nodemask 做“与”运算来确定叶节点位置，最后$\mathbf{leafindex}[T_h]$中第一个为 1 的比特的位置即为输入样本最终落入的叶节点位置。

若按照真假节点的定义来对每个输入数据逐个判断真假节点，其复杂度高达$O(\Lambda)$，Λ为叶节点个数，得不偿失。QuickScorer 采用了新的布局方式，它将所有决策树的所有节点按照特征的值的大小按照从小往大的顺序排列，如图 12.5 最上方 thresholds 的排列所示。对于每个输入数据而言，所有比它对应特征的值小的节点都是假节点，比它大的节点都是真节点。

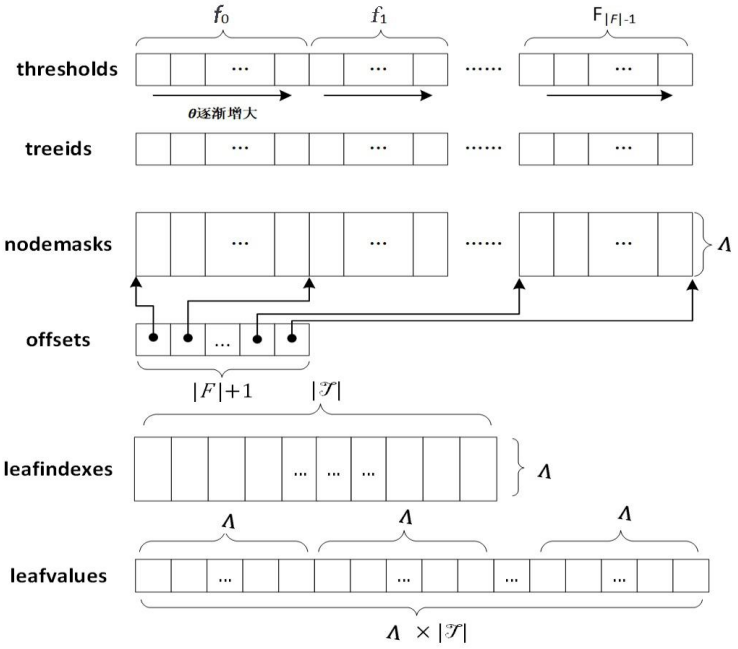


图 12.5　QuickScorer 的数据存储方式

所以，QuickScorer 的运算主要分成了三个部分：①假节点发现：对于输入数据，快速确定哪些节点是假节点；②假节点“与”运算：将假节点的 nodemask 进行“与”运算，得到 leafindexes 的结果；③叶节点位置计算：从 leafindexes 中快速地找出第一个 1 的位置，即为落入的叶节点位置，通过该位置得到对应的叶节点的值。所有树的预测结果累加起来即为该输入数据对应的最终答案。

12.3.2 RapidScorer 数据结构设计

本节我们将说明如何设计新的数据结构，能够最大程度地舍弃不必要的计算。

通过观察 QuickScorer 的执行过程，我们发现两个步骤存在不必要的计算：①QuickScorer 的节点表示方式使得它的计算复杂度与叶节点的个数紧密相关。当叶节点数目大于 64 时，一个“与”运算都需要多条运算指令才能完成，这样的计算代价会很高。②在决策树规模较大时，很多节点在特征和阈值上是相同的，可以合并，来减少不必要的计算。

1. 优化“与”运算操作

我们观察这些节点的 nodemask 表示可以发现，这些 nodemask 中真正参与运算的部分所占比例很小。因为，在位运算中，我们知道，对于任意比特 y，有性质$0 \wedge y = 0,\ 1 \wedge y = y$。即对于我们需要更新的$\mathbf{leafindex}[T_h]$来说，只有和比特 0 做“与”运算时，才会影响$\mathbf{leafindex}[T_h]$的值。譬如对$n_5$的 nodemask 1111101，只有一个比特参与了运算，即第 6 位的 0。

通过数据计算，我们可以发现，在平均情况下，比特 0 在每个 nodemask 中仅占$\frac{\sqrt{\pi\Lambda}}{2\Lambda}$，其中$\Lambda$为叶节点的个数。譬如，当 Λ 为 512 时，比特 0 只占 7.8%的比例。举个例子，在图 12.4 树节点表示中，我们可以看到节点n_3、n_4、n_5的 nodemask 中都只有一个 0；n_1、n_2只有两个 0。鉴于比特 0 在 nodemask 里只占很小的比例，依靠这个性质，假如设计一种结构，能够只用 nodemask 中含有 0 的部分做

运算，那会大大加速求解leafindex[T_h]的速度。

为此，我们设计了一种四元组结构 ep={fb, fbp, eb, ebp}来代替 nodemask。其中，fb 表示第一个含有 0 的字节；fbp 表示 fb 的位置；eb 表示最后一个含有 0 的字节；ebp 表示 eb 的位置。在这里，有读者可能好奇，为什么我们将字节(byte) 作为最小单位，而并没有用比特 (bit) 作为最小单位。其原因是，我们希望能够使用 SIMD 来进一步优化执行速度。这部分内容，会在 12.3.3 节阐述。有了 ep 这个四元组结构，就可以重新改写leafindex[T_h]与nodemask的“与”运算：

```
1: function EPITOME_AND(ep, leafindex[T_h]):
2:     leafindex[T_h][ep.fbp] ← leafindex[T_h][ep.fbp] ∧ ep.fb
3:     leafindex[T_h][ep.fbp + 1 : ep.ebp] ← 00...0 bytes
4:     leafindex[T_h][ep.ebp] ← leafindex[T_h][ep.ebp] ∧ ep.eb
5:     return leafindex[T_h]
```

通过 Epitome_AND，我们大大减少了不必要的运算。进一步，我们设计一种二元组结构 epS={fb, fbp}，epS 是 ep 的简化版，用在一个字节就能覆盖所有 0 的情况。这样的情况非常普遍，很多靠近叶节点的内部节点的 nodemask 都满足这种情况。我们统计过，即使叶节点到 400 时，还有 66%的节点可以被一个字节覆盖。epS 不仅能简化存储，也能减少运算，它只需要运行函数 Epitome_AND 的第 2 行语句就可以。有读者可能会有疑问，那是不是需要一个额外的判断语句来判断是否执行第 3 行和第 4 行呢？我们的答案是——不用。因为我们将 epS 的节点和 ep 的节点分开存放，各自运算。

2. 等效节点合并

在 RapidScorer 中，我们将特征相同、阈值相同的节点称为等效节点，将这些等效节点合并也能够减少很多比较运算。GBDT 含有多棵决策树，每棵决策树有不少节点，其中，许多节点的特征和阈值都相同，这些节点原来很可能分布在不同的决策树上。但是，QuickScorer 的内部节点排列方式却提供了一种优化的可能性：QuickScorer 提供了一种按照特征及阈值的内部节点排列方式，在这种

方式下，原先属于不同树的特征和阈值相同的节点被排列在了一起，如图 12.6 所示。对于这些节点，如果我们能把它们合并，仅做一次比较运算来确定是否是假节点，那么能够节省检测假节点的计算量。

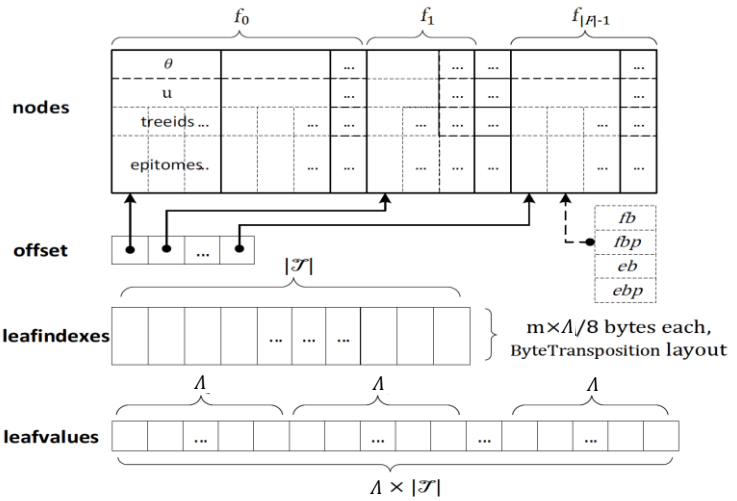


图 12.6　RapidScorer 的数据存储方式

为了做这些特征和阈值相同的节点的合并，我们设计了一种结构 eqnode =(θ, u, treeids, epitomes)，其中 θ 是阈值，u 是合并的节点的数目，treeids 是每一个节点对应的决策树的编号，epitomes 是每一个节点的 nodemask 的 epitome 表示，于是有了图 12.6 的 RapidScorer 数据存储方式。有了 eqnode 结构，我们只需要一次比较就能判断这些节点是否是假节点。

12.3.3　RapidScorer 矢量化

为了使用 SIMD 矢量化强大的加速能力，我们使用 SIMD 对多个数据输入的同时计算，譬如 AVX，就能同时处理 32 字节，考虑到我们的 epitome 格式，也就是能够支持 32 个数据的同时计算。本节的目标就是将 RapidScorer 算法的计算矢量化，做到每一步计算都能够使用 SIMD 指令来执行。这样，就能最大程度

地利用计算机的计算效率。

现在，我们使用 SIMD 来优化 12.3.1 节中提到的 3 个步骤。

步骤一：假节点发现。

原方法：在 QuickScorer 中，真假节点的判断由一行判断指令完成：$x_k \leqslant p.\theta$（p 为第 i 个内部节点 n_i，θ 为该节点阈值）。若判断结果为真，则为真节点；若为假，则为假节点。

SIMD 优化：在 RapidScorer 中，使用 SIMD 可以同时判断一个节点对多个样本真假节点的情况：$\vec{\eta} \leftarrow \overrightarrow{x_k} \leqslant \overleftarrow{p.\theta}$。其中真假节点信息记录到 $\vec{\eta}$ 中，我们用正向的箭头表示有多个元素的向量，譬如 $\vec{\eta}$，$\overrightarrow{x_k}$；特殊地，如果这些元素都是相同的，我们用反向箭头表示，譬如 $\overleftarrow{p.\theta}$。记录的真假节点信息 $\vec{\eta}$ 对于接下来的步骤二有很大的帮助。

步骤二：假节点“与”运算。

原方法：在 QuickScorer 中，$\text{leafindex}[T_h]$ 的更新通过与假节点的 nodemask 做“与”运算完成，即：$\text{leafindex}[T_h] \longleftarrow \text{leafindex}[T_h] \wedge p.\text{nodemask}$

SIMD 优化：在 RapidScorer 中，我们提出了 epitome 的节点表示方式，并实现了 Epitome_AND()函数。现在，我们通过 SIMD 实现它的向量化版本 Vectorized_AND()，它能同时处理多个样本。

```
1: function VECTORIZED_AND(p, leafindex[T_h]⃗, η⃗):
2:     leafindex[T_h][p.fbp]⃗ ← leafindex[T_h][p.fbp]⃗ ∧ (η⃗ ∨ p.fb⃖)
3:     if p.fbp ≠ p.ebp then
4:         for k = p.fbp + 1 to p.ebp − 1 do
5:             leafindex[T_h][k]⃗ ← η⃗ ∧ leafindex[T_h][k]⃗
6:         leafindex[T_h][p.ebp]⃗ ← leafindex[T_h][p.ebp]⃗ ∧ (η⃗ ∨ p.eb⃖)
```

SIMD 优化基于 epitome 表示，与 Epitome_AND()函数不同的是，Vectorized_AND()运算中还使用了步骤一中的结果向量 $\vec{\eta}$，来记录当前节点 p 对输入数据为真节点或假节点的情况。因为我们对多个数据同时处理，节点 p 对一

些输入数据而言是假节点，需要进行“与”运算；对另一些输入数据而言是真节点，不需要进行“与”运算。有了$\vec{\eta}$之后对于真节点，111…111 与任何二进制数做“或”运算还是 111…111 本身，其与二进制数进行“与”运算还是该二进制数，所以$\text{leafindex}[T_h] \wedge (111\cdots111 \vee p.\text{fb}) = \text{leafindex}[T_h]$。类似地，有$\text{leafindex}[T_h] \wedge (000\cdots000 \vee p.\text{fb}) = \text{leafindex}[T_h] \wedge p.\text{fb}$。

在设计 Vetcorized_AND()函数时，我们遇到的另一个困难是数据的存储方式。我们需要保证 SIMD 处理的数据是在内存中连续存放，这样 SIMD 指令能一次性读入这些数据，才能更好地利用 SIMD 的加速性能。为此，我们在设计数据结构时，将原本对不同样本的$\text{leafindex}[T_h]$以逐个排列的方式做了改变，变成对应的字节放在一块儿，如图 12.7 最上方的表格所示，不同样本的$\text{leafindex}[T_h]$的同位置的字节在内存中连续存放，同时读入 SIMD 寄存器中进行操作。

步骤三：叶节点位置计算。

原方法：在 QuickScorer 中并没有明确指出它们采用了何种方式来快速计算叶节点。不过，从位运算的角度分析，这其实是一个计算先导零数目的过程。我们翻阅了一些位操作的资料，最后参考了斯坦福大学的位操作来实现。不过，斯坦福大学的位操作实现计算的是后导零的数量，也就是从右往左连续的零的数量。我们需要计算的是从左往右连续的零的数量。看到这里，相信读者已经有想法了：既然数据结构是我们自己定义的，那我们不妨将计算方式也改成计算后导零，即，将从右往左的第 1 位来对应第一个叶子节点，第 2 位对应第 2 个叶子节点，以此类推。这样我们不就可以应用斯坦福大学的位操作来实现了吗？是的，我们依照这个想法实现了从叶子节点从 8、16、32……512 个的快速计算叶节点的方法。以下为叶节点为 64 时，即$\text{leafindex}[T_h]$的宽度为 64 时快速计算叶节点的函数实现：

```
   static const UInt32 Mod67BitPosition[] = // map a bit value mod 67 to its position,
table of finding the position leftmost bit "1"
```

```
{
    64,0,1,39,2,15,40,23,3,12,
    16,59,41,19,24,54,4,0,13,10,
    17,62,60,28,42,30,20,51,25,44,
    55,47,5,32,0,38,14,22,11,58,
    18,53,63,9,61,27,29,50,43,46,
    31,37,21,57,52,8,26,49,45,36,
    56,7,48,35,6,34,33
};

#define findIndexLeftmost1(val) \
    Mod67BitPosition[(-(val) & (val)) % 67]
```

这个函数本质上是一个哈希运算的过程，其中巧妙的是-(val) & (val)这个运算，其结果仅保留 val 二进制下最低位 1，即，假设 val 有 1 的最低位为第k位，则-(val) & (val)结果为2^k。与 67 取模的运算实质上是为了做一个哈希映射，将稀疏的2^k的输出映射到 0~66 的范围。选择 67 的原因是 67 是大于 64 的最小素数。Mod67BitPosition[]记录的是2^k在 67 上映射之后k的值。

RapidScorer 方法：findIndexLeftmost1()这个方法确实很高效，但是在对多个数据的同时处理时不能完全采用 SIMD 指令来并行。我们在本节步骤二中提到，每一个$\mathbf{leafindex}[T_h]$是按字节拆开排列，将它们重新组合成$\mathbf{leafindex}[T_h]$势必浪费时间，而且在$\mathbf{leafindex}[T_h]$的长度很长时，一个长整数类型并不能足够表示$\mathbf{leafindex}[T_h]$。考虑到存储方式的实际特点，我们提出了一种新的快速计算所有数据对应的叶节点位置的方法，即，先按字节寻找，对每个数据寻找不为 0 的第一个字节，记录字节位置；然后从字节中寻找第一个不为 0 的比特的位置。最后所有数据的叶节点位置即为字节位置乘以 8 加上比特位置，其过程如图 12.7 所示。

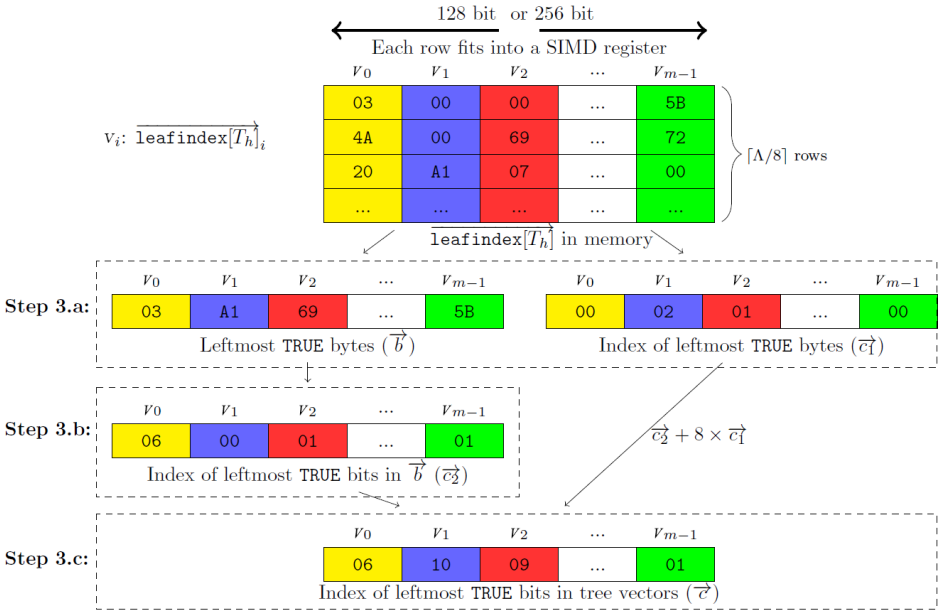


图 12.7　RapidScorer 计算叶节点位置的方式

我们来看一个例子，图 12.7 中红色一列，v_2的 16 进制表示为(00, 69, 07…)。我们的算法是先看第一个字节 00，其为 0，于是继续看第二个字节 69，其非 0，于是将字节 69 记录到$\vec{b}$中，该字节的位置 01 则记录到$\vec{c_1}$中。然后我们再对$\vec{b}$中寻找第一个不为 0 的比特的位置，其为 01。于是最终的位置即为 01×8+01=09。这里有读者可能会有疑问，当有的数据已经找到对应的字节或者比特，而有的数据还没找到时，怎么办？我们的做法是，继续往后扫描，直到所有数据都找到对应的字节或者比特。为此，我们会有一个额外的 SIMD 向量，来记录每个数据是否已经找打对应的字节或者比特。

12.3.4　RapidScorer 实验结果

通过多种优化，我们的 RapidScorer 取得了很好的加速效果。我们在开放数据集 MSN 和必应广告数据集 AdsCTR 上分别做了实验，将 RapidScorer 与其他类似算法进行比较：QuickScorer、V-QuickScorer、VPred、XGBoost。实验

环境是 Intel Core E5-1650 v4，时钟频率 3.6Ghz，内存 32GB RAM，三级缓存分别是 32KB、256KB 和 15MB，实验结果如表 12.1 所示。

表 12.1　不同加速算法对样本平均处理时间的实验结果（单位：ms）

算法比较	叶节点数	决策树数量							
		1,000		5,000		10,000		20,000	
		MSN	AdsCTR	MSN	AdsCTR	MSN	AdsCTR	MSN	AdsCTR
RapidScorer(AVX)	8	0.9 (-)	1.4 (-)	4.6 (-)	6.3 (-)	9.1 (-)	11.8 (-)	18.1 (-)	23.1 (-)
RapidScorer(SSE)		1.1 (1.2x)	1.8 (1.3x)	5.2 (1.1x)	7.6 (1.2x)	10.3 (1.1x)	13.5 (1.1x)	20.6 (1.1x)	26.4 (1.1x)
QuickScorer		2.4 (2.7x)	4.9 (3.4x)	11.1 (2.4x)	18.6 (3.0x)	22.2 (2.4x)	36.0 (3.0x)	44.8 (2.5x)	64.5 (2.8x)
VPred		7.8 (8.8x)	8.9 (6.2x)	38.9 (8.4x)	44.4 (7.1x)	76.0 (8.3x)	88.8 (7.5x)	153.3 (8.4x)	177.2 (7.7x)
XGBoost		25.9 (28.9x)	30.9 (21.5x)	122.7 (26.4x)	178.9 (28.5x)	261.1 (28.6x)	359.9 (30.4x)	604.6 (33.3x)	760.9 (32.9x)
RapidScorer(AVX)	16	1.4 (-)	2.1 (-)	6.8 (-)	9.5 (-)	12.8 (-)	17.6 (-)	21.7 (-)	35.1 (-)
RapidScorer(SSE)		1.5 (1.1x)	2.6 (1.3x)	7.8 (1.1x)	11.5 (1.2x)	15.0 (1.2x)	22.0 (1.2x)	24.5 (1.1x)	43.3 (1.2x)
QuickScorer		3.1 (2.2x)	7.5 (3.6x)	16.5 (2.4x)	29.0 (3.1x)	33.2 (2.6x)	53.7 (3.0x)	73.6 (3.4x)	105.9 (3.0x)
VPred		16.2 (11.4x)	16.0 (7.6x)	80.3 (11.8x)	79.8 (8.4x)	161.5 (12.7x)	160.5 (9.1x)	319.7 (14.7x)	326.8 (9.3x)
XGBoost		38.3 (27.1x)	49.4 (23.5x)	185.4 (27.2x)	236.2 (24.9x)	423.6 (33.2x)	475.0 (26.9x)	921.8 (42.5x)	976.5 (27.9x)
RapidScorer(AVX)	32	2.5 (-)	3.4 (-)	11.9 (-)	15.2 (-)	25.0 (-)	30.4 (-)	48.3 (-)	58.8 (-)
RapidScorer(SSE)		2.7 (1.1x)	4.1 (1.2x)	13.4 (1.1x)	17.9 (1.2x)	28.4 (1.1x)	35.6 (1.2x)	54.4 (1.1x)	69.9 (1.2x)
V-QuickScorer(AVX)		3.2 (1.3x)	5.1 (1.5x)	17.8 (1.5x)	23.0 (1.5x)	39.3 (1.6x)	49.4 (1.6x)	81.0 (1.7x)	110.0 (1.9x)
QuickScorer		5.1 (2.1x)	11.5 (3.4x)	27.2 (2.3x)	49.0 (3.2x)	62.6 (2.5x)	94.6 (3.1x)	168.6 (3.5x)	229.5 (3.9x)
VPred		31.2 (12.6x)	30.7 (9.0x)	164.9 (13.8x)	158.4 (10.4x)	345.8 (13.8x)	332.1 (10.9x)	720.8 (14.9x)	736.3 (12.5x)
XGBoost		47.8 (19.4x)	63.7 (18.7x)	264.9 (22.2x)	303.8 (20.0x)	560.8 (22.4x)	636.9 (20.9x)	1343.9 (27.8x)	1253.1 (21.3x)
RapidScorer(AVX)	64	4.3 (-)	6.0 (-)	20.8 (-)	28.2 (-)	40.2 (-)	56.8 (-)	88.1 (-)	124.8 (-)
RapidScorer(SSE)		5.3 (1.3x)	6.8 (1.1x)	25.4 (1.2x)	32.0 (1.1x)	45.8 (1.1x)	64.7 (1.1x)	98.6 (1.1x)	141.2 (1.1x)
V-QuickScorer(AVX)		7.3 (1.7x)	12.8 (2.1x)	40.2 (1.9x)	61.4 (2.2x)	139.6 (3.5x)	144.1 (2.5x)	304.2 (3.5x)	361.9 (2.9x)
QuickScorer		10.4 (2.4x)	19.7 (3.3x)	57.8 (2.8x)	99.1 (3.5x)	154.6 (3.8x)	220.7 (3.9x)	435.9 (4.9x)	514.5 (4.1x)
VPred		59.9 (14.0x)	56.5 (9.4x)	338.7 (16.3x)	320.0 (11.3x)	736.4 (18.3x)	683.0 (12.0x)	1284.8 (14.6x)	1372.8 (11.0x)
XGBoost		60.6 (14.2x)	78.2 (13.0x)	366.3 (17.6x)	387.1 (13.7x)	821.1 (20.4x)	778.3 (13.7x)	2280.3 (25.9x)	1899.2 (15.2x)
RapidScorer(AVX)	128	6.8 (-)	11.0 (-)	33.8 (-)	55.5 (-)	75.5 (-)	124.2 (-)	230.5 (-)	366.9 (-)
RapidScorer(SSE)		8.9 (1.3x)	12.4 (1.1x)	43.6 (1.3x)	62.2 (1.1x)	96.5 (1.3x)	139.7 (1.1x)	290.2 (1.3x)	415.0 (1.1x)
QuickScorer		39.2 (5.8x)	64.4 (5.8x)	268.4 (7.9x)	430.5 (7.8x)	618.9 (8.2x)	1086.9 (8.8x)	2083.6 (9.0x)	3143.8 (8.6x)
VPred		77.6 (11.5x)	79.5 (7.2x)	441.8 (13.1x)	436.9 (7.9x)	921.3 (12.2x)	952.2 (7.7x)	2154.3 (9.3x)	2114.1 (5.8x)
XGBoost		85.0 (12.6x)	89.8 (8.1x)	515.4 (15.2x)	504.1 (9.1x)	1156.0 (15.3x)	926.1 (7.5x)	3112.4 (13.5x)	2805.5 (7.6x)
RapidScorer(AVX)	256	12.3 (-)	21.2 (-)	70.1 (-)	119.1 (-)	184.7 (-)	282.3 (-)	595.8 (-)	793.1 (-)
RapidScorer(SSE)		16.4 (1.3x)	24.1 (1.1x)	88.5 (1.3x)	139.2 (1.2x)	228.2 (1.2x)	341.1 (1.2x)	710.1 (1.2x)	913.6 (1.2x)
QuickScorer		148.8 (12.1x)	182.2 (8.6x)	1226.9 (17.5x)	1817.7 (15.3x)	3167.0 (17.1x)	4276.2 (15.2x)	6937.5 (11.6x)	9042.6 (11.4x)
VPred		91.0 (7.4x)	96.7 (4.6x)	529.9 (7.6x)	558.6 (4.7x)	1203.7 (6.5x)	1264.9 (4.5x)	2645.9 (4.4x)	2715.1 (3.4x)
XGBoost		117.5 (9.6x)	101.0 (4.8x)	675.4 (9.6x)	524.6 (4.4x)	1356.9 (7.3x)	1268.6 (4.5x)	4570.0 (7.7x)	3934.2 (5.0x)
RapidScorer(AVX)	400	25.2 (-)	36.0 (-)	196.8 (-)	269.0 (-)	475.2 (-)	768.0 (-)	1260.6 (-)	1799.2 (-)
RapidScorer(SSE)		34.0 (1.3x)	41.6 (1.2x)	240.8 (1.2x)	319.5 (1.2x)	581.5 (1.2x)	887.5 (1.2x)	1438.0 (1.1x)	2001.0 (1.1x)
QuickScorer		536.0 (21.3x)	727.0 (20.2x)	4847.1 (24.6x)	6231.3 (23.2x)	11858 (25.0x)	13198 (17.2x)	23465 (18.6x)	28628 (15.9x)
VPred		123.4 (4.9x)	125.6 (3.5x)	729.6 (3.7x)	743.7 (2.8x)	1798.0 (3.8x)	1953.4 (2.5x)	3915.8 (3.1x)	4130.5 (2.3x)
XGBoost		139.1 (5.5x)	111.6 (3.1x)	868.1 (4.4x)	977.0 (3.6x)	2070.5 (4.4x)	2006.4 (2.6x)	5125.6 (4.1x)	4943.9 (2.7x)

在实验中，我们大跨度地考虑了 GBDT 模型的多种情况，叶节点规模从 8、16 到 400，决策树的规模从 1000、5000 到 20000。表 12.1 中记录了不同加速算法对样本平均的处理时间，橙色为处理速度最快的算法，蓝色为除 RapidScorer 算法外处理速度最快的算法。可以看到，用 AVX 实现的 RapidScorer 在所有情况中表现最好，远超过其优化的基准——QuickScorer。

12.4　本章小结

本章介绍了一些优化执行效率的方法，分别是近似计算、矩阵运算加速、多线程处理，以及优化数据结构等。这些优化应用在了 Seq2Seq 和 GBDT 的加速

中，并取得了良好的效果。实际上，在运算层面，不同的深度学习算法大同小异。类似的矩阵运算优化和指数函数计算优化可以推广应用到大多数深度学习算法的运行优化上。

参考文献

[1] HOCHREITER S, SCHMIDHUBER J, Long short-term memory. Neural computation[J]. 1997, 9(8):1735-80.

Deep Learning Model and Application

第 13 章

深度学习的下一个浪潮

通过对前面 12 章内容的学习，相信读者已经对深度学习的基本概念、重要算法和应用场景有了一定的了解。本章将对深度学习的未来做一些展望，希望对致力于这一领域学习和发展的读者有所帮助。

13.1 深度学习的探索方向展望

从深度学习算法模型的角度，我们认为下面的几个方面将是研究人员探索的主要方向。

13.1.1 设计更好的生成模型

生成对抗网络已经成为深度学习领域里备受关注的一类模型框架。得益于其简单巧妙的框架结构和灵活多样的模型选择，研究人员可以设计出各具特色的生成对抗网络用于多种不同的应用场景。尤其是对于带有创作性质的应用，比如作诗、写对联、作词、作曲、绘画等，已经有很多示范性的工作出现。然而，就整体的效果而言，目前基于生成对抗网络生成模型的表现总体来说还处于一般水平，需要研究人员投入更多的精力，去设计和改进出可以学习如何生成人眼无法分辨的图像、语音和文本的算法，从而使得这些生成模型能够达到产品级别的应用水平。

13.1.2 深度强化学习的发展

Google DeepMind 研发的智能围棋程序 AlphaGo 一举掀起了人工智能的新浪潮，对于在大众中普及人工智能知识起到了非常积极的作用。AlphaGo 成功的核心算法 Deep Q-Network（DQN）就是强化学习与深度学习相结合的产物。近年来，围绕 DQN 算法，研究人员已经把深度强化学习从游戏扩展到了很多更加实际的应用中。比如，搜索引擎、计算广告系统、推荐系统都可以被看作是智能体（agent），从这些系统与用户的交互行为中可以定义出状态、动作、奖赏，

并通过深度强化学习来训练出最优的推荐策略或者排序策略。但是目前这些技术都还处于尝试阶段，还有相当大的空间可以改进提高。另外，在人们普遍感兴趣的机器人控制领域，深度强化学习大有可为，是研究人员一直在花费大量精力探索的一个重要方向。

13.1.3 半监督学习与深度学习

半监督学习可以在只有少量标注数据的大数据集上进行学习，利用标注数据和未标注数据的相关联性和分布规律等性质发现并使用一些潜在的标注信息，从而提升学习性能。我们可以把很多半监督学习算法与深度学习进行结合，从而设计出一些针对不同问题的新算法。这一方向也将是有可能会出现突破性进展的研究领域。

13.1.4 深度学习自身的学习

目前深度学习的网络结构大多是人们事先定义好的，人们对于不同结构的优劣和使用范围也有了一些经验性的认识。在设计好神经网络的结构以后，模型训练中的调参数又是一个非常烦琐冗长的过程。在理想的状况下，深度学习算法应该能够实现自身架构的学习和调整，并且具有自动调整自身超参数等功能。在这个领域中研究人员已经有了一些初步的探索，但是距离目标还非常遥远。我们期待在不久的将来，对于网络结构的自动调整和自动调参会有更多的研究成果面世，因为这将把深度学习推向一个崭新的高度。

13.1.5 迁移学习与深度学习的结合

深度学习一般是在大量数据的基础上训练一个复杂的神经网络来完成特定的任务。但是有时会遇到这样的情况，对于某个问题我们并不具备充分的数据，但是对于和它很相似的另一个问题却有充足的数据。例如，我们有大量的英语文本数据及它们的分类标签，但是荷兰语的文本只有少量数据和标签，就可以先通

过深度学习训练一个高质量的英语文本分类器，然后根据两种语言数据的关系通过迁移学习来训练一个高质量的荷兰语文本分类器。通过迁移学习，我们可以解决以前难以处理的小数据问题和个性化问题。我们只需要找到与小数据问题和个性化问题有关联的一个大数据问题，就可以通过深度学习与迁移学习的结合来解决它们。

13.1.6 用于推理的深度学习

目前比较成熟的深度学习领域主要是特征学习方面，即算法可以在大量训练数据中自己学习出有效的特征，而不用像以前一样主要依靠人工手动设计和挑选特征。这样一来，算法可以在更大的搜索空间里找到最合适的一批特征。但是，对于把深度学习用于推理的研究还处于初步阶段。这一方面的研究，需要把深度学习与知识图谱、逻辑推理、符号学习相结合。如果在这一方面有所突破，将会对深度学习甚至人工智能的发展起到强大的推进作用。

13.1.7 深度学习工具的标准化

从第 2 章我们可以知道，当前深度学习的工具有很多种可以选择，各大公司都会开源和推广它们的计算平台和工具包，每隔一两年也都会有一两种新的工具面世，每个工具都有自己的特点。我们预计若干年后，深度学习领域将建立起一组核心标准化工具框架，从而更方便各类开源工具的传播和应用。对于深度学习从业者而言，这样的标准化工具框架也有利于知识的积累、比较、更新，并可以加速创新和研发的进程。

13.2 深度学习的应用场景展望

13.2.1 医疗健康领域

在医疗健康领域，深度学习可以应用于疾病诊断、病患状态分析、病情咨询、

药物研发、精准医疗等方面。

1. 疾病诊断

以癌症为例，对于大多数类型的癌症来说，活体组织检查（简称活检）是医生诊断癌症的主要方式。其他的一些测试（比如血液中某些成分指标的测定）可能表明癌症是存在的，但是只有活检可以做出明确的诊断。活检是指从患者体中取出少量组织在显微镜之下进行检查，由医生来辨别患者体内是否有肿瘤细胞，如果有，那么是良性肿瘤还是恶性肿瘤。医生主要通过对病变组织及细胞形态的分析、识别，再结合肉眼观察及临床相关资料，做出各种疾病的诊断。这个过程一般需要几天时间，而且对医生的系统知识和经验要求较高。

活检其实是可以通过医学影像识别（主要用到计算机视觉中的图像目标检测和分类）来完成的。由于每年都有大量患者被诊断为癌症，医疗机构已经积累了大量的相关医学数据。对于每次活检，样本在显微镜下的图像、医生标出的肿瘤细胞（包括良性和恶性）、患者的其他检验报告（比如血液检查）、医生的诊断结果等都完整地保存了下来。对于某一种类型的癌症，如果我们收集到了大量的活检数据，就可以使用深度学习中的各种卷积神经网络模型来对样本图像进行肿瘤细胞的目标检测、分类判定等训练，并把训练出来的模型用于活检的初检，供医生做最终诊断的参考，从而大大降低医生的工作量并提高诊断效率。

除用于癌症诊断的活检外，核磁共振成像（MRI）设备产生的医学图像处理和诊断是深度学习的另一个重要应用领域。在这个领域中，美国 Arterys 公司旗下的深度学习解决方案 Arterys Cardio DL 在 2017 年年初获得了美国食品及药物管理局（FDA）的批准认可（见图 13.1）。

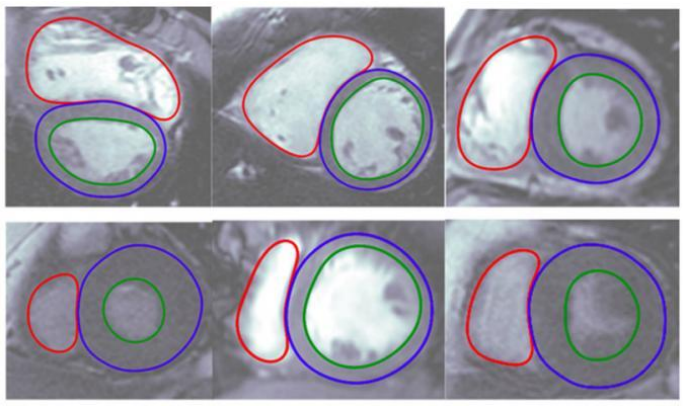


图 13.1　Arterys Cardio DL 对核磁共振成像的处理结果，图中展示了其为心脏 MRI 提供的自动化、可编辑的心室分割图像结果，为医生诊断提供帮助

2．病患状态分析

在病人被诊断出某种疾病，医生开出治疗方案以后，病人就进入治疗阶段。在这个阶段，病人的状态跟踪和治疗干预对于治疗效果而言非常重要。对于重症患者一般需要住院治疗，患者的一些身体指标如心电图、呼吸状况、体温等会实时收集到护士站进行监控，患者也要定期做一些血液化验等检查，医生可以随时调整治疗方案。对于一般患者，医生会给出处方进行药物治疗，然后每隔一段时间患者需要回到医院进行复查，医生根据检查结果判定治疗完成或者调整治疗方案。

病患状态分析和自动处理也是可以通过机器学习算法来实现的，比如深度强化学习模型。对于患者的治疗过程，一般医生都会留有病历。病历上会记载患者的病情状况、检验结果、医生的诊断及治疗方案，在复诊时会记载治疗效果、复检结果，以及后续治疗方案。对于重症病人，病历的内容更加丰富，甚至包括专家会诊意见等信息。对于某一种疾病，如果我们收集了大量的病历数据，就可以通过深度强化学习来训练出一个病患状态分析模型。我们可以把病患的病情和检验结果作为状态，把医生的治疗方案作为动作，把复查时的疗效作为奖赏，那么我们就可以把大量病历所反映的治疗情况时序数据作为训练样本来学习一个深

度强化学习模型。模型训练好以后，它就可以根据新病人的状态信息给出一些策略动作（治疗方案），供医生参考。对于很多治疗情况良好或者处于正常恢复阶段的病患，可以大大减轻医生的分析和决策精力，从而医生可以把更多的精力用于疑难杂症和危重患者的治疗。

在这方面，已经有研究人员进行了探索，比如图 13.2 展示了 2018 年发表的一篇应用深度强化学习中的动作评价网络来对病人治疗方案的动态调整进行推荐的工作。

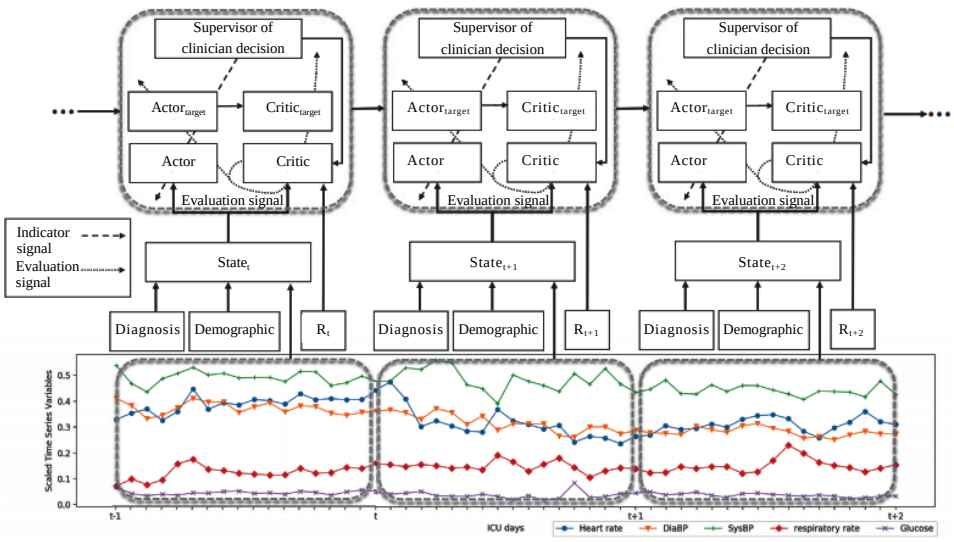


图 13.2　用于治疗方案动态调整推荐的动作评价网络

3. 病情咨询

病人在治疗过程中往往会有一些疑惑，比如有些疼痛在减轻了几天以后又严重起来、服用的一些药物导致了不良反应、疗程过半就感觉已经痊愈了等。这时病人最需要的是主治医生的判断和一些简洁的回复，但是如果病人为此再跑去医院挂号排队就诊则十分耗费精力，医生也不得不分出一些时间来处理这些不十分紧急的情况，降低了医疗资源的使用效率。

在医疗机构积累了大量病患治疗资料后，我们可以基于这些资料来建立一个智能问答聊天助手来解决这类患者的问题。关于智能聊天助手，目前苹果有 Siri，Google 有 Google Now，微软有小娜，亚马逊有 Alexa，阿里巴巴有天猫精灵。这些智能聊天助手一般是基于机器学习的一些框架，比如搜索引擎、推荐系统、问答系统等。在这些领域里面到处都有深度学习的影子，比如基于词向量的语句表示方法、基于序列到序列模型的智能问答系统等。在建立了病情咨询聊天助手以后，患者可以通过和聊天助手的互动来快速了解病情发展、解除疑惑，或者采取行动（比如提前回医院复查）。这样既可以缓解病患压力提高治疗效果，也可以提升医疗资源的使用效率。在这方面，也已经有了一些尝试，比如 MedWhat 就可以回答一些疾病的成因、治疗方法、诊断方式等。

4. 药物研发

一般来说，药物研发的周期长、成本高、风险大。目前已经有些制药公司开始引入深度学习的模型算法来辅助药物研发。比如通过基于卷积神经网络的医学图像处理系统，科研人员可以在分子层面研究人体健康组织、探究人体细胞自身防御组织，以及发病机制等，然后利用基于深度学习的回归预测模型来推算与人体自身分子相匹配的潜在药物化合物。这种方法不仅可以大大降低新药研发的时间和成本，还使得靶向治疗成为当今医学治疗的新趋势。在这一方面的发展，生物科技公司 Berg 研发的 Interrogative Biology 系统和 IBM 研发的 Watson Health 系统都创造了十分良好的开端。

5. 精准医疗

提到了靶向治疗，就不得不提近年来越来越受关注的精准医疗。精准医疗的核心技术是对人类基因组的测序。通过对人类基因组的测序和研究，科研人员就可以从病理根源来诊断疾病，并确定相应的靶向治疗方案。但是人类基因组十分庞大，而且包含大量进化中残留的无用信息，还包含一些较短的重复序列。其中包含的有用信息也具有不同的功能属性，十分复杂。如何在庞大的人类基因组信

息里找到对于某种疾病的治疗有影响的片段是一个非常有挑战性的工作。深度学习中的递归神经网络是处理序列信息的重要工具，科研人员正在尝试用它来进行基因序列分析。我们期待不远的将来在这方面有革命性的成果出现。

13.2.2 安全隐私领域

在安全隐私领域，深度学习可以应用于智能安防系统和智能行为分析系统等方面。

1. 智能安防系统

在智能安防领域，深度学习应用最广泛的是计算机视觉方向，即对图像和视频的检测和分析。在传统的安防领域，在部署了大量的监控设备之后，对图像和视频的检测和分析主要靠人。比如在一些大型建筑的监控室里，通常可以看到满墙的监视器阵列，安保人员 24 小时不间断地盯着监视器观察异常情况。在出现安保状况以后，调查人员需要把某段时间某几个监控点的视频资料调取出来，然后用快进的方式进行人眼线性扫描从而找出异常片段。这种工作方式效率低而且耗费大量人力，况且人总有疲惫走神的时候，可能会漏掉一些异常情况。

近年来，深度学习在计算机视觉方面进展飞速，在对图像和视频的人脸识别、物体检测、目标分类等任务中都取得了非常接近人类甚至超越人类的识别准确率。这些技术都已经逐渐应用到了各类大型安防系统中，完成诸如行人检测、车辆检测、非机动车检测等任务。随着深度学习算法的改进和加速，智能安防系统也逐步进入了千家万户。比如，Nest 家庭安防系统，在主人离家时智能摄像头可以检测到家里的异常是有人进入还是有物体移动，并能够识别出宠物的行动从而避免误报警。图 13.3 展示了 Nest 家庭安防系统提供的一系列智能硬件产品。

图 13.3 Nest 家庭安防系统系列产品

2. 智能行为分析系统

为了给广大居民提供生活上的便利，各大银行、商家、快递公司会把很多自动提款机、自动售货机、自助存取快递柜等设备放置在居民小区、商场、写字楼等区域。这些设备也需要安全防护，从而保护公私财产，便利居民使用。目前，已经有专为这些自动设备研发的智能行为分析系统面世。这种系统除了具备人脸检测和识别的功能之外，还能自动分析操作自助设备的人的行为，可以对加装设备、粘贴虚假告示、破坏设备等行为进行判断和报警。

13.2.3 城市治理领域

在城市治理领域，深度学习可以应用于智能交通控制、环境指标预测、人员密度预测等方面。

1. 智能交通控制

城市交通拥堵问题已然成为城市治理中的顽疾，很多大城市都会出台限牌、限行、收取拥堵费等措施来减缓交通拥堵，但是随着城市化进程的加速，人们越来越多的交通需求与各种限制措施的矛盾日益加深，拥堵使得城市生活品质下降、效率降低。除进行更加科学的城市规划、道路改造等措施外，优化改造现有的交通控制系统是耗资少、周期短、见效快的一种方案。现有的交通信号灯系统一般都具有基本的配置功能，可以人工设置不同时间段内各个方向信号灯的时长，但是这种设置完全需要人来根据经验进行调整。一个大城市有成千上万个路口，每个路口不同时段不同季节的交通流量都会有不同的规律，时而发生的交通事故也会对周围几个路口的交通流量产生影响。因此，这种人工设置的方式是非常低效率的。

智能交通控制系统可以通过布置在路口的几个监控摄像头来实时获取每个方向的交通流量，运用深度学习中图像和视频检测识别的方法对机动车、非机动车、行人进行识别、追踪和计数，然后实时控制交通信号灯的切换，避免某个方向已经没有车辆通过但是仍然长时间放绿灯的情况出现，从而提高路口使用效

率，缓解拥堵。

2. 环境指标预测

随着生活水平的提高，人们对生活品质的要求也越来越高，比如近年来十分受关注的空气质量问题，很多人每天早上都要查看空气质量预报来决定当天出行是否需要戴口罩等防护措施。空气质量预测有一定的难度，一般来说一个城市的空气质量监测点只有几十个，而且分布比较分散。而城市中某个街区的空气质量受到地形、交通、风力、风向的影响而变化，很多情况下实际数据和附近的空气质量检测点的数据有一些差距。为了可以对城市里每个街区的空气质量进行实时预测，有科研人员已经开始运用基于深度学习的预测模型，并取得了不错的预测结果。

3. 人员密度预测

人员密度预测是现代城市治理中的一项重要任务，及时地预测到某些区域的人员密度，发现人员聚集地、判别聚集原因并疏散人群，对于城市管理者而言非常重要。这有助于缓解交通、防止踩踏故事，以及及时发现群体性事件等，其目的都是提高城市安全保障城市秩序。很多所谓智慧城市项目就运用基于深度学习的视频检测、各类信息整合等技术，来对人员密度陡增的区域进行预警，城市管理者可以迅速派出工作人员进行调查和疏解，来保障正常的城市秩序。

以上这些有关城市治理的应用基本上都是各种“智慧城市”项目的重要组成部分，比如阿里云研发的城市大脑项目。

13.2.4 艺术创作领域

在艺术创作领域，深度学习可以应用于作诗、作曲、绘画等，主要用于娱乐消遣性的活动。在艺术创作领域，人工智能是不能完全取代人类的。艺术创作需要人的感性认知、灵感闪现等特性，不是人工智能程序可以轻易替代的。但是如果只为单纯的娱乐消遣，用程序自动生成一些诗词歌赋、音乐、美术作品，也是

可以丰富人们的生活并增加交流乐趣的。

我们知道，已经有很多研究工作运用生成对抗网络来产生各种不同艺术风格的美术作品，例如梵高风格的太空船和莫奈风格的电脑。当然他们两位都没有画过这些东西，但是这样的作品很有趣，可以博人一笑。微软亚洲研究院多年前曾经推出过一款叫“微软对联”的程序，可以根据用户给出的上联来自动生成下联和横批，如图 13.4 所示。

图 13.4　微软对联示例

近年来，腾讯和阿里巴巴逢年过节的时候都推出过一些作诗和作对联的小应用，用户可以输入几个关键字来自动生成有趣的作品，然后通过社交软件分享给亲朋好友，增加节日气氛。微软的智能聊天产品微软小冰甚至出版了一本原创诗集《阳光失了玻璃窗》（见图 13.5），其背后运用的是输入图片输出文字的 Image2Text 技术。上面这些工作都或多或少地用到了自然语言处理中的一些深度学习模型。我们从《阳光失了玻璃窗》里摘了一首智能程序生成的诗，大家可以感受一下。

我的爱人哪

快把光明的灯擎起来了

那里有美丽的天

问着村里的水流的声音

我的爱人在哪

因为我的红灯是这样的幻变

像是美丽的秘密

她是一个小孩子的歌唱

那时间的距离

图 13.5　人工智能创造的诗集《阳光失了玻璃窗》封面

13.2.5　金融保险领域

在金融保险领域，深度学习可以应用于收益预测、投资组合构建、风险管理、产品推荐等方面。

1. 收益预测

对于金融产品的收益预测，传统的方法一般是基于时间序列分析，比如自回

归滑动平均模型（Autoregressive Integrated Moving Average，ARIMA）和向量自回归模型（Vector Autoregression，VAR）。如果引入深度学习算法，我们可以使用深度回归模型、卷积神经网络以及长短期记忆网络来处理金融产品的时序数据。此外，对于金融产品的收益研究，传统方法还包括对产品相关新闻报道、政策文件等背景资料的分析。在深度学习的框架下，这些背景资料可以很方便地通过各种文本分析的方式融合到深度预测模型中，从而取得更好的预测效果。

2. 投资组合构建

投资组合构建是一个组合优化和收益预测相结合的问题，除传统的组合优化方法外，研究人员已经开始尝试用自动编码器（Auto-Encoder）等深度神经网络模型来进行建模。具体来讲，可以用自动编码器将时间序列映射到自身，然后使自动编码器的预测误差成为股票测试版的代用指标，训练好的自动编码器可以用作真实市场的预测模型来帮助构建投资组合。

3. 风险管理

风险管理是金融行业里面的一个关键问题，对于金融机构的资金安全起着重要的保障作用。金融机构可以根据客户的一些画像特征、历史行为数据等抽取特征，然后用深度学习算法训练信用分数回归模型或者信用评级分类模型，从而确保贷款可以发放给信用良好的客户。在贷款拨付以后，金融机构可以继续跟踪客户的资金使用、偿还等行为，抽取相应的特征，并运用深度学习算法对客户未来的偿还能力进行预估，对于有风险的客户可以提前进行介入，进而保障资金安全。在这一方面，研究人员已经进行了很多探索，比如有的使用图 13.6 所示的深度神经网络来对客户的信用进行评级，还有使用图 13.7 所示的深度神经网络来根据客户行为进行欺诈交易检测（比如信用卡的盗刷等）。

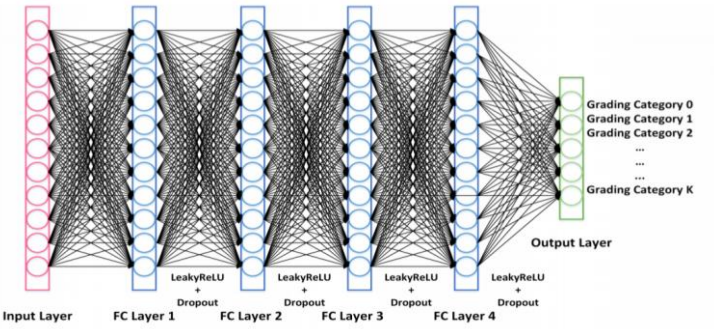


图 13.6　用于信用评级的深度神经网络

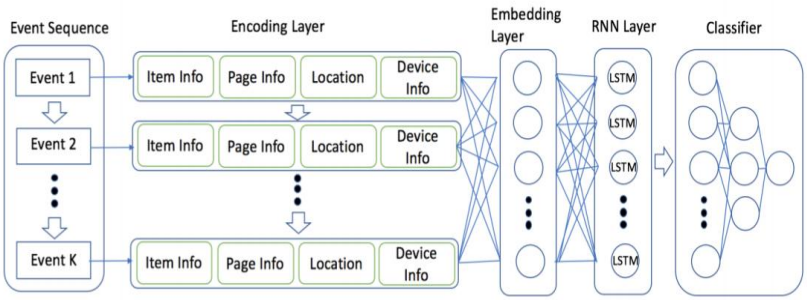


图 13.7　用于欺诈交易检测的深度神经网络

4. 产品推荐

金融产品推荐需要根据客户的投资偏好和风险承受能力来进行仔细筛选，通常都是经过理财顾问与客户的细致沟通之后才能做出相对好的推荐。这一方面，我们其实可以参考网上商城的商品推荐算法，根据客户的画像信息（年龄、职业、收入、资产状况等）、客户在金融机构的历史行为信息（贷款、投资、理财等）作为输入，通过基于深度学习的推荐算法在银行的智能手机 APP 中对客户进行自动而且精准的金融产品推荐，从而销售更多的金融产品。

13.2.6　无人服务领域

在无人服务领域，深度学习可以应用于无人车、无人店、机器人等方面。这些应用比较吸引眼球，相信读者已经在新闻报道中有一定的了解。

1. 无人车

无人车是智能汽车的一种，也称为自动驾驶汽车或者轮式移动机器人，主要依靠车内的基于计算机系统的智能驾驶系统来实现无人驾驶的目的。智能驾驶系统中有多个功能模块都是基于深度学习的。比如，车身周围安装有多个采集图像的摄像头，这些图像需要通过图像分析算法来识别道路标志线、信号灯、交通提示牌，并且要识别和跟踪周围的车辆和行人；汽车的语音识别系统需要能理解乘车人发出的语音指令（比如更改目的地、临时靠边停车等），并把指令传送给智能驾驶系统；汽车的自动驾驶策略可以通过深度强化学习来训练。总之，无人车的智能驾驶系统是多个方面的深度学习系统的整合，随着技术的进一步发展和完善，无人车将会对人类出行方式产生革命性的影响。图 13.8 展示了多个公司正在研发的无人车项目。

2. 无人店

无人店指商店内所有或部分经营流程，通过技术手段进行智能化自动化处理，从而降低或彻底取消人工干预。这一领域最受关注的是亚马逊推出的无人便利店 Amazon Go（见图 13.9）。Amazon Go 颠覆了传统便利店、超市的运营模式，使用基于深度学习的计算机视觉技术以及传感器技术等，彻底跳过传统收银结账的过程。顾客进入 Amazon Go 的时候需要扫描一下其智能手机里的 Amazon Go 应用程序，这样系统就知道哪些顾客正在店里选购商品。Amazon Go 的货架可以感知人与货架的相对位置以及货架上商品的移动，再结合商店天花板上的多个摄像头捕捉到的图像的人体识别和购物袋识别来判断是谁拿走了哪个商品。当顾客完成商品选择之后，就可以直接离开 Amazon Go 了。稍后 Amazon Go 的结算系统就会把账单发送给顾客的 Amazon Go 应用程序并从其关联的银行卡里完成扣款。目前 Amazon Go 无人店还处于尝试阶段，当店内人多拥挤的时候或者多人同时在某个方向取商品的时候，图像识别算法会出现很多误判，这需要从设计或者算法层面进行不断的改进。我们期望未来不久，无人店的相关图像识别技术能有突破性进展，在提高识别准确性的同时可以降低硬件需求从而节省成本，

这样无人店就可以被广泛推广和使用了。

（a）Google 的无人车

（b）特斯拉的无人车

（c）百度的无人车

（d）福特的无人车

图 13.8　Google、特斯拉、百度、福特正在研发的无人车

图 13.9　Amazon Go 无人店

3. 机器人

广义来讲，机器人是自动执行工作的机器装置。它既可以接受人类指挥，又可以运行预先编排的程序，还可以根据基于人工智能的算法行动。机器人工业涵盖的范围十分庞大，有很多专业的资料介绍，我们就不一一展开了。近年来比较受关注的机器人项目是波士顿动力（Boston dynamics）研发的各种仿生机器人（见图 13.10），其中的机器狗动作敏捷，仿生程度非常高，人形机器人可以像人一样熟练地单腿连续跳上堆叠起来的箱子，非常震撼。总之，机器人所涉及的计算机视觉、语音识别、自然语言处理、智能问答、回归预测、行动策略学习等领域都可以看到深度学习的身影，这里面每一项技术的革新都可以触发机器人工业的进步。

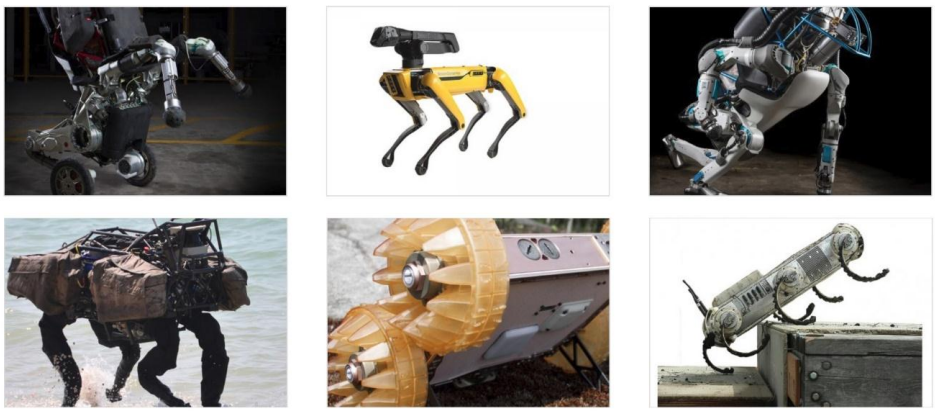


图 13.10　波士顿动力研发的各种仿生机器人

13.3　本章小结

本章我们对深度学习的未来发展方向进行了一些展望，主要包括生成模型的改进、深度强化学习的发展、半监督学习与深度学习的结合，以及深度学习自身的学习等几个方面。我们还对未来深度学习的应用场景做了一些展望，主要在医

疗健康、安全隐私、城市治理、艺术创作、金融保险、无人服务等几个方面。希望通过本章内容的介绍，能够使读者开阔眼界，使读者认识到深度学习的前景非常广阔，在这一领域不论是进行科学研究还是产业应用都有希望做出非常有影响力的工作，来改变我们的生活，改变我们的世界。

参考文献

[1] LECUN Y, BENGIO Y, HINTON G. Deep learning. Nature [J] 521 (7553):436,2015.

[2] CHOLLET F. Deep Learning with Python, Manning Publications, 2017.

[3] WANG L, ZHANG W, HE X, et al, Supervised reinforcement learning with recurrent neural network for dynamic treatment recommendation. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining 2018[C]. ACM,2447-2456.

[4] TAN F, HOU X, ZHANG J, et al, A novel risk assessment scheme and practice for peer-to-peer lending. Proceedings of ACM SIGKDD Workshop on Data Science of Fintech 2018.

[5] ZHANG R, ZHENG F, MIN W. Sequential Behavioral Data Processing Using Deep Learning and the Markov Transition Field in Online Fraud Detection. arXiv preprint arXiv:1808.05329,2018.